Markt&Technik

Eberhard Zehendner

Das Z80-BUCH

Assembler-Datenstrukturen-Programmaufbau

Für 8-Bit-Computer wie Schneider CPC, Joyce, MSX und Commodore 128.

Das Z80-Buch

Eberhard Zehendner

Das Z80-BUCH

Assembler - Datenstrukturen - Programmaufbau

Für 8-Bit-Computer wie
Schneider CPC, Joyce, MSX und Commodore 128

Markt & Technik Verlag AG

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Zehendner, Eberhard:
Das Z-80-Buch : Assembler, Datenstrukturen, Programmaufbau / Eberhard Zehendner. –
Haar bei München : Verlag Markt u. Technik, 1987.
ISBN 3-89090-219-7

15 14 13 12 11 10 9 8 7 6 5 4 3 2
90 89 88 87

ISBN 3-89090-219-7

Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Jantsch, Günzburg
Printed in Germany

Inhaltsverzeichnis

Vorwort

So mancher wird sich gedacht haben »schon wieder ein Z80-Buch, und nun gar erst **das** Z80-Buch«. Und in der Tat gibt es ja ein großes Angebot an solchen Druckwerken. Was also ist das Besondere an **diesem** Buch? Dazu ein kleiner Rückblick:

Um gut Programmieren zu lernen, braucht ein interessierter und geschickter Mensch nicht unbedingt ein gutes Buch. Um sich aber in kurzer Zeit, mit minimalem Aufwand und ohne größere Genieleistungen die Systematik des Programmierens anzueignen, ist dieses unverzichtbar (zumindest beim Erlernen der ersten Programmiersprache beziehungsweise Assemblersprache). Als ich im Sommer '85 ein Z80-Buch für meine Studenten suchte, wurde mir schnell klar, daß die auf dem Markt befindlichen Bücher eher den Charakter von Nachschlagewerken für Fortgeschrittene als den von systematischen Lehrbüchern besitzen. Was mir jedoch vorschwebte, war ein problemorientierter Ansatz, der den Befehlsvorrat des Prozessors nur als Mittel sieht, um Probleme der Praxis zu bewältigen. Primär kam es mir auf den Zusammenhang zwischen Problem, Lösungsweg und Programm an, und nicht auf eine möglichst vollständige (und am Ende auch noch alphabetische) Auflistung der Befehle des Z80. Also beschloß ich, selbst ein geeignetes Buch zu schreiben. Hier ist es!

Daß in diesem Buch trotzdem alle Befehle des Z80 auftauchen, ist das Verdienst der Entwickler von ZILOG, die den Z80 ausschließlich mit solchen Befehlen versehen haben, die man in irgendeinem Zusammenhang mit Vorteil verwendet. Und daß im Anhang des Buches auch eine alphabetische Liste der Befehle mit ihren Wirkungen (nebst weiterer, andersgeordneter Übersichten) erscheint, dürfte sich von selbst verstehen.

Nun zum Aufbau des Buches:

In den Kapiteln 1 bis 5 werden grundlegende Dinge erklärt, die mit der Programmierung des Z80 zu tun haben. Kapitel 1 dient im wesentlichen der Begriffsbestimmung und Motivationsanalyse. Wer bereits fit ist im Rechnen mit binären, oktalen und hexadezimalen Zahlen, kann Kapitel 2 ohne weiteres überschlagen (aber wie wär's mit einer kleinen Kontrolle? Dieses Kapitel enthält Übungsbeispiele, mit Lösungen am Ende des Buchs).

In Kapitel 3 sind die verwendeten Methoden zur Beschreibung von Programmabläufen zusammengestellt. Den Z80 bekommen wir erstmals in Kapitel 4 zu sehen. Schließlich wird in

Kapitel 5 erklärt, wie Programme entwickelt, erstellt, übersetzt, lauffähig gemacht und getestet werden.

In den Kapiteln 6 bis 25 werden wichtige Datenstrukturen und praktisch alle möglichen Ablaufstrukturen (Kontrollmöglichkeiten wie Verzweigung, Schleife, Unterprogramm) dargestellt. Die einzelnen Kapitel bauen häufig aufeinander auf, Daten- und Ablaufstrukturen bedingen sich teilweise gegenseitig. Der Programmierneuling sollte diese Kapitel deshalb der Reihe nach durcharbeiten. Wer schon Z80-Erfahrung besitzt, wird diesen Teil gern als Fundus für Standardlösungen benutzen.

In Kapitel 26 bis 28 werden schwierige, erfahrungsgemäß kritische Probleme der Programmierung behandelt: Ein-/Ausgabe, Unterbrechungen und verschiebbare Programme. Nur für Geübte!

Im letzten Kapitel möchte ich noch einige besonders schöne Programmierbeispiele erklären, die für die Praxis relevant sind.

Kapitel 2 sowie Kapitel 6 bis 28 enthalten Übungsaufgaben, an Hand derer der Leser seine Kenntnisse überprüfen kann. Die Lösungen (oder besser gesagt Lösungsvorschläge, denn es gibt meist viele Lösungen) befinden sich unmittelbar anschließend an Kapitel 29.

Ein ausführlicher Anhang hilft bei Fragen zu Befehlssatz und Assembler-Pseudo-Operationen. Er enthält auch Angaben über Code-Länge und Ausführungszeiten.

Die Beispiele sind so gewählt, daß sich die verwendeten Methoden meist wiederholen; beim ersten Lesen braucht deshalb nicht alles gleich bis ins letzte Detail verstanden werden. Auch das Verständnis der vielen Begriffe des ersten Kapitels stellt sich mit fortschreitender Praxis schnell ein; für die ersten Schritte in die Welt des Programmierens sind die meisten Begriffe nicht besonders wichtig.

Noch eine Bemerkung zum Schluß: Fast jedes Buch enthält Fehler, und Programmierbücher gehören dabei zu den schlimmsten. Also bitte nicht böse sein, wenn auch in diesem Buch der eine oder andere Fehler im Text, oder schlimmer, im Programm auftaucht, denn »nobody is perfect«. Über Hinweise dazu und über Anregungen zum Inhalt freue ich mich immer!

Daß ich dieses Buch schreiben konnte, verdanke ich Herrn Professor Dr. Hans-Joachim Töpfer. Bei der Arbeit bin ich von meinen Korrekturassistenten – besonders von Peter Moll – tatkräftig unterstützt worden.

Mein Dank gilt auch dem Verlag Markt & Technik, insbesondere meinem Lektor, ohne die **das Z80-Buch** nicht hätte erscheinen können.

Der Autor

1
Einführung

In diesem ersten Kapitel sollen zunächst einmal wichtige Begriffe im Zusammenhang mit Assemblerprogrammierung geklärt werden. Die Bedeutung und Anwendungsbereiche der Assemblerprogrammierung werden ebenso genannt wie spezielle Vorteile des Prozessors Z80. Es wird der prinzipielle Aufbau eines Mikroprozessors skizziert. Der Einordnung des Z80 in die große Gruppe der Mikroprozessoren dient ein Abschnitt über Klassifizierung.

1.1 Begriffsbestimmung

Höhere Programmiersprachen sind Sprachen, die dem Benutzer eine Menge von Daten- und Ablaufstrukturen zur Verfügung stellen, welche dem Benutzerproblem wesentlich näher stehen als der Maschinenstruktur.

Maschinennahe Programmierung arbeitet dagegen mit Sprachen, die lediglich eine verständliche Syntax über die eigentliche Maschinensprache legen, sonst aber die benutzten Strukturen eins zu eins in die Maschinensprache abbilden. Die maschinennahe Programmierung wird in zwei Formen praktiziert: als *Assemblerprogrammierung* und als *Mikroprogrammierung.*

Bei der *Assemblerprogrammierung* befindet sich das Programm in einem externen Speicher und wird - ein Befehl nach dem anderen - zur Verarbeitung in den Prozessor geholt. Jeder Assemblerbefehl entspricht im wesentlichen einem Maschinenbefehl.

Ein *Mikroprogramm* realisiert einen Maschinenbefehl des Prozessors (sozusagen die Feinstruktur des Befehls). Der Mikroprogrammspeicher ist logisch gesehen ein Teil des Prozessors (selbst wenn er nicht im Prozessor untergebracht ist).

Prinzipielle Unterschiede zwischen Assemblerprogrammierung und Mikroprogrammierung bestehen darüber hinaus nicht.

1.2 Grundbegriffe der maschinennahen Programmierung

Ein *Computer* besteht aus den Komponenten *Prozessor*, *Speicher* und *Ein-/Ausgabe-Geräte*.

Die kleinste in einem Computer isolierbare *Informationsmenge* kann genau zwei Ausprägungen haben (wahr - falsch); diese Informationsmenge wurde als ein *Bit* definiert. Zur Aufbewahrung eines Bit benötigt man ein *Bit* Speicher (ein Bit ist eine - in heutigen Computern meist elektronische - Speichervorrichtung, die genau zwei verschiedene Zustände annehmen kann: an - aus, oder auch: hohes Potential - niedriges Potential). In den meisten Speichermedien, insbesondere denen von Mikrocomputern, sind je 8 Bits zu einer größeren Einheit zusammengefaßt, dem *Byte* (ein Byte kann im Prinzip auch eine andere Anzahl von Bits enthalten, dies ist aber nicht sehr gebräuchlich; wir werden deshalb stets annehmen, daß ein Byte 8 Bits besitzt). 2 Bytes können wiederum zu einem Wort zusammengefaßt werden.

Jedes Byte eines Speichers ist durch eine *Adresse* ansprechbar. Eine Adresse ist eine vorzeichenlose ganze Zahl. Die Adressen eines Speichers werden meist fortlaufend von 0 ab gezählt.

Innerhalb eines Bytes sind die Bits von 0 bis 7 numeriert, innerhalb eines Worts von 0 bis 15. Bei der Darstellung von Zahlen entspricht heute üblicherweise Bit 0 der letzten (niederwertigsten) Binärziffer. Man nennt dann Bit 0 das *LSB* (least significant bit), während Bit 7 des Bytes beziehungsweise Bit 15 des Worts *MSB* (most significant bit) heißt. Entsprechend (und sehr zur Begriffsverwirrung beitragend!) heißt innerhalb eines Worts das Byte mit der niedrigeren Adresse *LSB* (least significant byte), das mit der höheren Adresse *MSB* (most significant byte). Wir werden stets von dieser Wertigkeit der Bits beziehungsweise Bytes ausgehen, wenn wir numerische Daten darstellen wollen.

Aus der Sicht des Prozessors ist der Speicher des Computers ein externer Speicher, im Gegensatz zum internen Speicher, der sich im Prozessor selbst befindet. Der interne Speicher ist organisatorisch partitioniert; die einzelnen Teileinheiten nennt man *Register*. Entsprechend der Speicherkapazität eines Registers (8 Bit, 16 Bit, ...) spricht man von 8-Bit-Registern, 16-Bit-Registern, ...

Der externe Speicher ist durch einen *Datenbus* und einen *Adreßbus* mit dem Prozessor verbunden. Über den Datenbus werden Datenwerte zwischen Speicher und Prozessor ausgetauscht. Die Adressierung der entsprechenden Speicherzellen erfolgt durch den Adreßbus.

Mit den Ein-/Ausgabe-Geräten kommuniziert der Prozessor durch sogenannte *Ports*. Jedem Gerät sind dabei ein oder mehrere Ports fest zugeordnet. Die Ports werden ähnlich wie die Speicherzellen durch eine *Portadresse* angesprochen. Die Daten werden ebenfalls auf einem Datenbus zwischen Prozessor und Ein-/Ausgabe-Geräten transportiert; dieser Datenbus kann mit dem des Speichers identisch sein.

Ein Verfahren, das eine bestimmte Aufgabe durch eine festgelegte Folge von Verfahrensschritten löst (oder zu lösen versucht), heißt *Algorithmus* (Plural Algorithmen). *Programme* sind die Realisierung von Algorithmen.

Assemblerprogramme werden mit Hilfe eines *Editors* als *Quell-Programme* erstellt. Ein *Assembler* genanntes Dienstprogramm übersetzt das Quell-Programm entweder direkt in die *Maschinensprache* (den sogenannten *Objekt-Code*) oder in einen *Zwischencode*; ein anderes Dienstprogramm, der *Binder* (engl. linker), fügt den Zwischencode aller zu einem kompletten Programm gehörenden Teilprogramme (Module) zu einem umfassenden Zwischencode

zusammen. Ein *Relokator* genanntes Dienstprogramm erzeugt dann schließlich daraus den Objekt-Code. Zum Objekt-Code gehören stets auch die *Datenbereiche*, auf denen der Code arbeitet (die Konstanten und Variablen). Ein *Macro-Assembler* ist ein Assembler, der spezielle Funktionen zur Textexpansion und Textsubstitution für die Entwicklung von Programmen zur Verfügung stellt.

Objekt-Code oder Zwischencode wird auch von den *Compilern* für höhere Programmiersprachen generiert; dem Objekt-Code kann man nicht mehr ansehen, ob er von einem Compiler oder einem Assembler erzeugt wurde.

Aufgabe des Relokators ist es, die im Zwischencode enthaltenen *relativen Adressen* (relativ zur späteren Anfangsadresse des Programms) in *absolute Speicheradressen* umzusetzen.

Der Objekt-Code eines Programms wird von einem sogenannten *Lader* (engl. loader) in den Hauptspeicher gebracht und gestartet. Manchmal sind die Funktionen des Binders und des Laders in einem einzigen Programm zusammengefaßt, dem *Binder-Lader* (engl. linking loader). Der Relokator kann ein eigenständiges Programm sein; meist ist er jedoch in den Binder oder in den Lader (zum Beispiel im UCSD-PASCAL) integriert.

Ein Assemblerprogramm kann sich nicht nur direkt auf die Hardware des Computers stützen, sondern auch von *Funktionen des Betriebssystems* Gebrauch machen; die Beschreibung dieser Zugriffsmöglichkeiten auf das Betriebssystem nennt man *Systemschnittstelle.*

Für die meisten Compiler steht ein sogenanntes *Laufzeitsystem* (engl. run time system) zur Verfügung, das dem Objekt-Programm weitere, allerdings meist sprachabhängige, Funktionen zur Verfügung stellt.

Es gibt bestimmte Funktionen, die in vielen Anwendungen in immer wieder der gleichen Art und Weise benutzt werden; diese kann man *getrennt assemblieren* und in einer *Programm-Bibliothek* (engl. library) ablegen. Der Binder sorgt dann dafür, daß die benötigten *vorübersetzten Unterprogramme* (engl. subroutines) aus der Bibliothek in den Objekt-Code eingebaut werden. Zur Verwaltung von Bibliotheken gibt es spezielle Dienstprogramme, *Bibliotheks-Verwalter* (engl. library manager) genannt.

Baut man Objekt-Programme aus sehr vielen verschiedenen Bibliotheken zusammen, so benötigt man zur *Dokumentation* und zur *Fehlersuche* ein Verzeichnis, das spezifiziert, welche Daten- und Programmadressen in welchem Unterprogramm und welcher Bibliothek definiert sind. Dies nennt man *Querverweise* (engl. cross reference).

Zum *Austesten* erstellter Programme und zur *Fehlersuche* benutzt man einen sogenannten *Debugger* (manchmal wird auch der Begriff *Monitor* verwendet). Mit dem Debugger kann man Speicherinhalte inspizieren, sich den Objekt-Code in einem bestimmten Speicherbereich auflisten lassen und den Ablauf eines Programms schrittweise (auch durch Simulation) verfolgen. Dabei können Speicher- und Registerinhalte sowie die ausgeführten Maschinenbefehle aufgezeichnet werden (engl. trace). Mit dem Debugger ist es auch möglich, den Objekt-Code eines Programms direkt zu verändern, ohne neu zu assemblieren oder zu compilieren (engl. patching).

Hat man ein Objekt-Programm, zu dem kein zugehöriges Quell-Programm in einer Assemblersprache existiert (zum Beispiel durch Compiler erzeugten Objekt-Code), so läßt sich dieses mit Hilfe eines *Disassemblers* in Assemblersprache rückübersetzen. Manchmal ist ein Disassembler im Debugger enthalten.

Zur Abarbeitung eines Programms besitzt der Prozessor ein Register, in dem sich stets die Adresse des nächsten Befehls befindet; dies ist der *Befehlszähler* (engl. program counter).

Die Ausführung eines Programms kann durch das *Eintreten äußerer Umstände unterbrochen* werden (engl. interrupt). Diese Umstände können Fehlersituationen sein wie zum Beispiel Stromausfall (engl. power fail) oder Abbau einer Wählverbindung (engl. disconnect), Benutzeraktionen wie Rücksetzen des Computers in einen Initialzustand (engl. cold boot) oder Eingabe von Daten über eine Tastatur, aber auch die Rückmeldung eines peripheren Bausteins von einem ihm erteilten separaten Auftrag. Bei Eintreffen eines entsprechenden Signals wird dabei erst ein bestimmtes Programm (engl. interrupt service routine) ausgeführt, bevor das unterbrochene Programm fortgesetzt wird.

Zur Erkennung bestimmter Ereignisse wie *Übertragungsfehler*, *Überlauf* oder *Division durch Null* besitzt der Prozessor spezielle Register, sogenannte *Flags*, die meist in einem *Flag-Register* zusammengefaßt sind.

Zur Realisierung von Unterprogrammsprüngen und Parameterübergabe bedient man sich häufig einer speziellen Datenstruktur, des *Stapels* (engl. stack). Dafür gibt es in vielen Prozessoren den *Stapelzeiger* (engl. stack pointer), das ist ein Register, welches die Adresse des obersten Stapel-Elements enthält.

1.3 Motivation für maschinennahe Programmierung

Die Entwicklung von Basis-Software für Computer erfolgt meist in mehreren Phasen, die sich hinsichtlich der Komplexität der verfügbaren Programmier-Umgebungen unterscheiden. Bevor Anwender-Programme in höheren Programmiersprachen erstellt werden können, müssen erst Dienstprogramme wie Editor und Compiler vorhanden sein; diese können entweder auf einem bereits bestehenden Computersystem in höheren Programmiersprachen entwickelt und dann auf das neue System transferiert werden (engl. cross compiling), oder sie werden direkt auf dem neuen System in einer Sprache programmiert, die - da ja noch kein Compiler verfügbar - notwendigerweise eine Assemblersprache sein wird. Diese Vorgehensweise, die mit einem kurzen, primitiven Monitor-Programm (mit dem wir einzelne Daten in den Speicher bringen können) beginnt und bei höheren Programmiersprachen wie zum Beispiel PASCAL endet, nennt man »boot-strapping«.

Höhere Programmiersprachen verbergen in der Regel die Vorgänge bei der Steuerung von Ein-/Ausgabe-Geräten. Die Ein-/Ausgabe wird von sogenannten »Treibern« erledigt; dies sind für ein bestimmtes Gerät spezifisch geschriebene Programme, die meist direkt auf den Hardware-Komponenten des Rechners operieren. Die Programmierung und Anpassung von Treibern erfolgt häufig in einer Assemblersprache.

Viele Mikrocomputersysteme besitzen überhaupt keine Compiler, sondern recht langsam arbeitende *Sprach-Interpreter* (typischerweise für die Programmierspache BASIC). An Programme dieser Interpreter können schnelle Assembler-Unterprogramme angebunden werden, die zeitkritische Funktionen (zum Beispiel Bildschirm-Steuerung) in Realzeit abwikkeln. Schnelle Ein-/Ausgabe-Programme sind darauf angewiesen, die Struktur der Maschine möglichst gut auszunutzen; dies ist nur durch maschinennahe Programmierung zu realisieren.

Bestimmte Programme (meist Systemprogramme) sind nur dann lauffähig, wenn Code und Daten unter fest vorgegebenen Adressen stehen. Compiler gestatten dem Benutzer häufig keinen Einfluß auf die Speicherlage; bei einem Assemblerprogramm bestimmt der Benutzer selbst die Ablageadressen.

Die genaue Kenntnis der Problem-Struktur (über die ein Compiler zwangsläufig nicht verfügt) kann der Programmierer ausnutzen, um den von einem Compiler erzeugten Code zu optimieren (oder gleich optimiert in Assembler zu schreiben); es gibt allerdings mittlerweile Compiler, die sehr effizienten Code erzeugen und so eine Nachoptimierung überflüssig machen.

»Patching«, das heißt Änderung bestehender Objekt-Programme, kann auch notwendig werden, wenn zum Beispiel

- keine Quellprogramme zu den Objekt-Programmen verfügbar sind, weil der Autor des Programms versucht, dadurch seine Urheber-Rechte zu wahren
- ein Objekt-Programm als Abbild des Speichers ohne zugrunde liegendes Quellprogramm erzeugt wurde
- eine Neuübersetzung der Quellen sehr zeitaufwendig ist (trifft typischerweise auf Updates von Betriebssystemen zu)
- in einem laufenden Programm ein Fehler entdeckt wird, der sofort behoben werden muß (On-Line-Systeme, zum Beispiel bei Banken)

Es darf nicht verschwiegen werden, daß unsachgemäßes Patching zur Zerstörung von Programm- und Datenbeständen führen kann, auch von solchen, die mit dem gepatchten Programm logisch gar nichts zu tun haben!

1.4 Prinzipieller Aufbau eines Mikroprozessors aus der Sicht des Programmierers

Vom Prozessor selbst sieht der Programmierer nur die Register. Mittels der Datentransportbefehle erhält er über Speicheradressen Zugriff auf die Inhalte des externen Hauptspeichers. Mittels der Ein-/Ausgabe-Befehle hat er über Portadressen Zugriff zu den peripheren Geräten (Tastatur, Bildschirm, Drucker, Plattenspeicher, Datenleitung). Somit kann man das Gesamtverhalten des Systems durch die Registerinhalte, Speicherinhalte und Zustände peripherer Geräte charakterisieren.

Dem Programmierer steht ein Vorrat von Maschinen-Befehlen zur Verfügung, die grob in folgende Funktionsklassen eingeteilt werden können:

- Datentransport zwischen den einzelnen Registern
- Datentransport zwischen Registern und externem Speicher
- Datentransport zwischen Registern und Ein/Ausgabe-Geräten
- Arithmetisch/logische Operationen auf den Registern
- Steuerung des Programmablaufs

Bei manchen Prozessoren kommen noch hinzu:

- Arithmetisch/logische Operationen auf dem externen Speicher
- Rotations- und Verschiebe-Operationen auf den Registern
- Rotations- und Verschiebe-Operationen auf dem externen Speicher

- Setzen, Rücksetzen und Testen einzelner Bits der Register
- Setzen, Rücksetzen und Testen einzelner Bits des externen Speichers

1.5 Klassifizierung von Mikroprozessoren

Eine erste Kenngröße ist die Breite des Daten-Busses; diese gibt an, wieviel Bit Information gleichzeitig von der Peripherie (Speicher und Ein-/Ausgabe-Geräte) zum Prozessor gebracht werden können. Die Breite des Daten-Busses variiert zur Zeit zwischen 1 Bit bis 32 Bit bei Mikroprozessoren; bei Prozessoren von Großrechnern sind 64 Bit und mehr keine Seltenheit.

Interessant ist ferner die Größe des adressierbaren Speichers (Adreßraum), die meist in Kilobyte (KB) oder Megabyte (MB) angegeben wird. Der Adreßraum wird bei manchen Prozessoren segmentiert, das heißt logisch in kleinere Teilstücke zerlegt.

Die Register eines Prozessors lassen sich einteilen in Spezialregister, die nur für spezifische Aufgaben zur Verfügung stehen, und multi-funktionale Register. Als Spezialregister ausgeführt sind normalerweise Segmentregister, Befehlszähler, Stapelzeiger, Unterbrechungs-Vektor-Register und die sogenannten »Flags«, die bestimmte Ereignisse wie zum Beispiel Überlauf anzeigen. Manche Prozessoren haben darüber hinaus noch Spezialregister für Datenadressierung, Arithmetik oder Schleifensteuerung.

Besitzt ein Prozessor relativ viele multi-funktionale Register, so sind diese meist in der Form eines Register-Arrays angelegt, das heißt sie sind alle gleich aufgebaut und werden durch einen Index bezeichnet, welcher der Speicheradresse eines externen Speichers entsprechen würde.

Von großem Interesse ist die Mächtigkeit des Befehlsvorrats, zusammen mit den Adressierungsmodi. Die Adressierungsmodi geben an, wie ein Operand spezifiziert werden kann (direkt, als Speicheradresse, als Registerinhalt, indiziert). Mächtige Befehlsvorräte beinhalten unter anderem Multiplikation und Division von ganzen Zahlen, arithmetische Operationen auf Gleitpunktzahlen, Zeichenketten-Verarbeitung und blockweisen Datentransport.

Häufig wird noch die Verarbeitungsgeschwindigkeit genannt, die man in Operationen pro Sekunde mißt. Diese Größe ist von zweifelhaftem Wert, da die verschiedenen Befehle eines Prozessors sehr unterschiedliche Ausführungszeiten besitzen können (vergleiche einen Multiplikationsbefehl mit einem einfachen Datentransportbefehl!); man behilft sich meist mit dem gewichteten Mittel aus den Ausführungszeiten, wobei die Gewichtung zu einer bestimmten Problemklasse abgeschätzt wird. Bei Auslegung eines Prozessors in verschiedenen Speichertechnologien ist die Taktfrequenz (gemessen in MHz) ein realistisches Maß für die Durchlaufzeiten der Befehle.

1.6 Gründe für die Wahl des Prozessors Z80

Bei der Wahl eines Prozessors zur Erlernung wesentlicher Techniken der Assemblerprogrammierung kommt es in erster Linie darauf an, dem Lernenden in kurzer Zeit ein solides Basis-Wissen zu vermitteln. Der Z80 ermöglicht dies durch seinen überschaubaren Befehlsvorrat, ohne gleichzeitig die Nachteile eines reinen *Lehr-Prozessors* zu besitzen.

Innerhalb der Klasse vergleichbarer Prozessoren (8-Bit-Datenbus, 16-Bit-Adreßbus) ist der Z80 relativ leistungsfähig.

Der Z80 ist weit verbreitet; er findet sich in Home-Computern, Personal-Computern, Interface-Karten und Peripheriegeräten wie zum Beispiel Druckern.

Für den Selbstbau eines Computersystems bietet sich der Z80 als billiger Prozessor an; als Grundsystem kann auch einer der vielen billigen Heim-Computer dienen (Schneider JOYCE zum Beispiel).

Der Z80 ist Teil einer Prozessor-Familie, die eine effiziente und sichere Verbindung des Prozessors mit der Peripherie gewährleistet.

Der Z80 wurde von der Firma ZILOG seit seiner ersten Freigabe mehrmals in seiner Arbeitsgeschwindigkeit wesentlich verbessert (Z80A, Z80B, Z80H). Außerdem gibt es mittlerweile Z80-kompatible Prozessoren, die den Befehlsvorrat des Z80 um mächtige Operationen wie zum Beispiel Multiplikation erweitern, auf denen aber beliebige Z80-Programme laufen können.

2
Darstellung ganzer Zahlen und Zahlsysteme

Wir sind es gewöhnt, Berechnungen stets im *Dezimalsystem* auszuführen, vielleicht weil wir zehn Finger haben. Dies bedeutet, daß wir Zahlen durch Folgen von *Dezimalziffern* darstellen, die sozusagen die *atomaren* Einheiten unseres Zahlsystems darstellen. Es gibt dabei genau *zehn* verschiedene Ziffern, weshalb man überhaupt erst von einem *dezimalen* System sprechen kann (lat. decem = zehn). Die einzelnen Ziffern einer bestimmten Zahl haben unterschiedliche Bedeutung: die letzte Ziffer der Zahl bedeutet einfach ihren Ziffernwert, zum Beispiel bedeutet die 3 eben »drei«, die vorletzte Ziffer bedeutet das Zehnfache ihres Ziffernwerts, zum Beispiel bedeutet die 2 dann »zwanzig«, und so fort. Der Gesamtwert der Ziffernfolge, also der Wert der Zahl, ist die Summe aller einzelnen, mit den entsprechenden Zehnerpotenzen *gewichteten* Werte der Ziffern. So ein Zahlsystem nennt man *Stellenwertsystem.* Ist eine Zahl d durch die Ziffernfolge $d_n...d_1d_0$ dargestellt, so hat d den Wert

$$\sum_{i=0}^{n} d_i * 10^i$$

(der Stern »*« steht für die Multiplikation, was bei Computern so üblich ist).

Ein Computer hat keine Finger, mit denen er bis 10 zählen könnte. Seine kleinste Speichereinheit ist das *Bit*, und dieses kann genau *zwei* Zustände annehmen: »an« oder »aus«, »gelöscht« oder »gesetzt«, »wahr« oder »falsch«, »Null« oder »Eins« (es gibt da noch mehr Interpretationen). Wir können daher nicht erwarten, daß ein Computer im Dezimalsystem arbeitet. Die meisten Computer sind aber so gebaut, daß zwischen unseren Rechengewohnheiten im Dezimalsystem und denen des Computers in seinem eigenen Zahlsystem gewisse Ähnlichkeiten bestehen, die es leicht machen, die Arithmetik des Computers zu verstehen.

2.1 Das Binärsystem

Die Zahlen des Computers sind ebenfalls aus Ziffern aufgebaut, nur sind es diesmal *Binärziffern*; das bedeutet, daß es genau *zwei* verschiedene Ziffern gibt, die »Null« und die »Eins« (lat. bis = zweimal). Das System, in dem Computer rechnen, heißt deshalb *Binärsystem.* Meist unterscheidet man in der Schreibweise nicht zwischen binärer und dezimaler Null beziehungsweise Eins, und stellt beide durch »0« beziehungsweise »1« dar.

Das Binärsystem ist ebenfalls ein Stellenwertsystem. Allerdings hätte bei nur zwei verschiedenen Ziffern eine Gewichtung mit Potenzen von 10 nicht viel Sinn. Im Binärsystem werden die Ziffern mit Potenzen von 2 gewichtet: die letzte Ziffer bedeutet ihren Ziffernwert, die vorletzte das Doppelte ihres Ziffernwerts, die vorvorletzte das Vierfache, und so weiter. Ist eine Zahl b durch die Ziffernfolge $b_n...b_1b_0$ dargestellt, so hat b den Wert

$$\sum_{i=0}^{n} b_i * 2^i.$$

Damit man Binärzahlen nicht mit Dezimalzahlen verwechselt, hängt man meist ein »B« an die Binärzahl an, etwa 101B für die (dezimale) Zahl 5. Für jemand, der noch nie im Binärsystem gerechnet hat, mag das alles ziemlich suspekt sein. Wir geben deshalb gleich einige Beispiele:

1010B = 10
1000B = 8
1111B = 15
11010010B = 210

Wir werden nun sehen, daß Berechnungen im Binärsystem fast genauso durchgeführt werden können wie im Dezimalsystem; sie sind in der Tat sogar etwas einfacher. Beginnen wir mit einem kleinen Beispiel:

Wir wollen die beiden Binärzahlen 111B (dezimal 7) und 10B (dezimal 2) addieren. Wir beginnen wie gewohnt bei der letzten Ziffer: 1 + 0 = 1. Als letzte Ziffer ergibt sich also »1«. Nun kommt die vorletzte Ziffer an die Reihe: 1 + 1 = 2, oder anders ausgedrückt: 1 + 1 = 10B. Wir erhalten also einen Übertrag zur drittletzten Stelle, als vorletzte Stelle ergibt sich »0«. Nun kommt die drittletzte Stelle zum Zuge, wobei wir jetzt den Übertrag einberechnen müssen: 1 + 0 + 1 = 10B. Die drittletzte Stelle des Ergebnisses lautet also »0«, und wieder erfolgt ein Übertrag zur nächsten Stelle. Die letzte Berechnung lautet also: 0 + 0 + 1 = 1. Insgesamt erhalten wir:

111B	7
+ 10B	+ 2
1001B	9

Es geht also alles wie gewohnt! Das Ergebnis, umgerechnet ins Dezimalsystem, stimmt mit unseren Erwartungen überein: 7 + 2 = 9 = 1001B. Hier noch zwei weitere Beispiele:

```
   110B      6        1111B     15
+ 10110B   + 22     +    1B   + 1
--------   ----     ---------  ---
  11100B     28       10000B    16
```

Das Subtrahieren von Binärzahlen funktioniert ähnlich einfach wie das Addieren. Wir geben auch dafür zwei Beispiele an (achte darauf, wie beim zweiten Beispiel das Borgen funktioniert):

```
  11011B     27       10110B     22
-  1010B   - 10     -  1001B   -  9
--------   ----     ---------  ---
  10001B     17        1101B     13
```

Besonders einfach ist das Multiplizieren im Binärsystem. Wer sich vielleicht noch mit Grausen an das Auswendiglernen des »kleinen Einmaleins« in der Schule erinnert, wird nun freudig überrascht sein; das »kleine Einmaleins« des Binärsystems lautet schlicht: $1 * 1 = 1$ (daneben muß man nur noch wissen, daß Null multipliziert mit einer beliebigen Zahl stets wieder Null ergibt). Das Vorgehen bei der Multiplikation dürfte aus folgenden beiden Beispielen hinreichend klar werden:

```
1011B * 1101B          111B * 10001B
------------           -------------
  1011                   111
  1011                   000
  0000                   000
  1011                   000
                         111
------------           -------------
10001111B              1110111B
```

Rechne die Beispiele selbst nach!

Bei der Division kommen wir im Binärsystem mit den Operationen *Vergleich* und *Subtraktion* aus. Zur Ermittlung einer einzelnen Ziffer des Ergebnisses prüfen wir, ob der Divisor größer ist als der zur Teilung anstehende Rest. Wenn ja, ergibt sich eine 0; wenn nein, so ergibt sich eine 1 und der Divisor wird subtrahiert. Nach dem Nachziehen der nächsten Ziffer des Dividenden wird der Vorgang wiederholt. Auch hierzu noch ein Beispiel:

```
  110100011B : 1010B = 101001B
- 1010
------
    1100
  - 1010
  ------
      10011
    - 1010
    ------
    Rest 1001
```

Was uns nun noch fehlt, ist eine Methode für die Umrechnung einer Dezimalzahl in die entsprechende Binärzahl. Wir geben dazu gleich zwei Methoden an (welche davon man als besser ansehen soll, ist Geschmackssache).

Die erste Methode macht von der Kenntnis der nötigen Zweierpotenzen Gebrauch. Wir bestimmen als erstes die größte Zweierpotenz, die nicht größer als die umzuwandelnde Zahl ist; dies sei zum Beispiel 2^n. Damit bekommt die Binärziffer b_n den Ziffernwert 1. Nun subtrahieren wir diese Zweierpotenz von der Zahl und wiederholen den Vorgang mit der entstehenden Zahl. Dies tun wir so lange, bis die Zahl auf Null zusammengeschrumpft ist. Diejenigen Binärziffern, die durch den Vorgang nicht den Wert 1 bekommen haben, erhalten nun abschließend alle den Wert 0.

Die ersten 16 Zweierpotenzen sind in folgender Tabelle zusammengefaßt:

Tabelle 2.1. *Zweierpotenzen*

n	2^n	n	2^n
0	1	8	256
1	2	9	512
2	4	10	1024
3	8	11	2048
4	16	12	4096
5	32	13	8192
6	64	14	16384
7	128	15	32768

Beispiel: $74 = 2^6 + 2^3 + 2^1$, also 74 = 1001010B.

Die zweite Methode baut die Binärzahl von rückwärts auf, beginnt also mit der letzten Binärziffer. Wir sehen uns dazu an, ob die Dezimalzahl gerade oder ungerade ist. Für eine gerade Zahl schreiben wir als Binärziffer b_0 eine 0, für eine ungerade Zahl eine 1. Dann halbieren wir die Dezimalzahl, wobei der anfallende Rest ebenfalls die gewünschte Binärziffer angibt. Nun wiederholen wir dieses Verfahren und bauen so nacheinander die Binärziffern $b_1, b_2, ..., b_n$ auf. Das Verfahren endet, wenn durch das fortgesetzte Halbieren die Dezimalzahl zu Null geworden ist. Auch hier ein Beispiel:

37 : 2 = 18 Rest 1
18 : 2 = 9 Rest 0
9 : 2 = 4 Rest 1
4 : 2 = 2 Rest 0
2 : 2 = 1 Rest 0
1 : 2 = 0 Rest 1

Also ergibt sich 37 = 100101B.

Übungen

1. Berechne im Binärsystem:

   1010B + 1010B = ?
   1110111B + 11011B = ?
   101000B − 1010B = ?
   11010111B − 1101011B = ?

2. Wandle folgende Dezimalzahlen ins Binärsystem um: 65, 127, 1194, 85.

3. Rechne die Beispiele aus Aufgabe 1 im Dezimalsystem nach!

2.2 Das Hexadezimalsystem

Das Operieren mit binär codierten Zahlen ist zwar einfach, aber umständlich. Viele Ziffern zu schreiben - im Binärsystem braucht man mehr als dreimal so viele Ziffern wie im Dezimalsystem - ist aufwendig, und man verschreibt sich leicht bei den langen Ketten von Nullen und Einsen. Deshalb faßt man gerne mehrere Binärziffern zu einer Einheit zusammen und codiert diese in ein anderes Stellenwertsystem um.

In einem *Stellenwertsystem* mit *Basis* B gibt es B verschiedene Ziffern (wenn B größer als 10 ist, nimmt man meist Buchstaben zu den Dezimalziffern hinzu). Ist eine Zahl z durch die Ziffernfolge $z_n...z_1z_0$ dargestellt, so besitzt sie den Wert

$$\sum_{i=0}^{n} z_i * B^i$$

Für unsere Zwecke gut geeignet ist das *Hexadezimalsystem* mit der Basis 16 (griech. hexa = sechs) oder das *Oktalsystem* mit der Basis 8 (lat. octo = acht). Beide besitzen nicht zu viele Ziffern und führen trotzdem schon zu hinreichend kurzen Ziffernfolgen.

Wir wollen uns zunächst mit dem Hexadezimalsystem befassen (statt *Hexadezimal* sagt man meistens kurz *Hex*). Das Hexadezimalsystem verfügt über die Ziffern 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Die Abbildung von je vier Binärziffern auf eine Hex-Ziffer (und umgekehrt) wird aus Tabelle 2.2. ersichtlich.

Hex-Zahlen werden meist durch ein nachgestelltes »H« gekennzeichnet, manchmal auch durch ein vorangestelltes »X«, »$«, »&« oder »'«. Einige Beispiele für die Umrechnung von Binärzahlen in Hex-Zahlen (und umgekehrt):

1010010101110B = 55EH
1000100111101001B = 12F9H
1100010110000001001B = CB09H

Tabelle 2.2. *Umrechnung von Hexadezimalziffern in Binärzahlen*

Hex	Binär	Hex	Binär
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Während Multiplikationen und Divisionen meist nicht im Hexadezimalsystem ausgeführt werden (dies wäre für uns doch zu ungewohnt, es sei denn, wir besitzen einen Hex-Taschenrechner!), bieten Addition und Subtraktion keine größeren Schwierigkeiten. Dazu je zwei Beispiele:

```
  1AH          801H
+  AH        + 1AC4H
-----        ------
  24H          22C5H

  10H          DCBAH
-  AH        - ABCDH
-----        ------
   6H          30EDH
```

Rechne alle vier Beispiele nach!

Übungen

1. Rechne folgende Binärzahlen ins Hexadezimalsystem um:
   11111111B, 101100111B, 1000001B, 11011B.

2. Wandle folgende Hex-Zahlen ins Binärsystem um:
   80H, C4H, 1234H, AAAAH.

3. Führe folgende Operationen im Hexadezimalsystem durch:
   36H + 14H = ?
   FFH - C3H = ?
   ADACH + E0FH = ?
   A000H - 0E32H = ?

2.3 Das Oktalsystem

Das Oktalsystem ist ein Stellenwertsystem mit der Basis 8. Es ist heute nicht mehr so gebräuchlich wie noch vor einigen Jahren und wurde insbesondere im Bereich der Mikrocomputer durch das Hexadezimalsystem abgelöst. Die Ziffern sind 0, 1, 2, 3, 4, 5, 6, 7. Je drei Binärziffern werden zu einer Oktalziffer zusammengefaßt. Die Zuordnung ist aus folgender Tabelle ersichtlich:

Tabelle 2.3. *Umrechnung von Oktalziffern in Binärzahlen*

Oktal	Binär	Oktal	Binär
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

Oktalzahlen werden meist durch ein nachgestelltes »O«, oder, weil dies leicht mit der Null verwechselt werden kann, durch ein nachgestelltes »Q« gekennzeichnet. Wir rechnen einige Binärzahlen in Oktalzahlen um:

11011B = 33Q
10111111B = 277Q

Hier noch je zwei Beispiele für Addition und Subtraktion im Oktalsystem:

3Q	64Q	167Q	227Q
+ 7Q	+ 36Q	− 63Q	− 153Q
12Q	122Q	104Q	54Q

Rechne alle Beispiele nach!

Sehr hilfreich beim Umgang mit den verschiedenen Zahlsystemen ist folgende Umrechnungstabelle:

Tabelle 2.4. *Umrechnung von Dezimal-, Hexadezimal- und Oktal-Zahlen*

Dezimal	Hex	Oktal	Dezimal	Hex	Oktal
000	00	000	003	03	003
001	01	001	004	04	004
002	02	002	005	05	005

Dezimal	Hex	Oktal	Dezimal	Hex	Oktal
006	06	006	046	2E	056
007	07	007	047	2F	057
008	08	010	048	30	060
009	09	011	049	31	061
010	0A	012	050	32	062
011	0B	013	051	33	063
012	0C	014	052	34	064
013	0D	015	053	35	065
014	0E	016	054	36	066
015	0F	017	055	37	067
016	10	020	056	38	070
017	11	021	057	39	071
018	12	022	058	3A	072
019	13	023	059	3B	073
020	14	024	060	3C	074
021	15	025	061	3D	075
022	16	026	062	3E	076
023	17	027	063	3F	077
024	18	030	064	40	100
025	19	031	065	41	101
026	1A	032	066	42	102
027	1B	033	067	43	103
028	1C	034	068	44	104
029	1D	035	069	45	105
030	1E	036	070	46	106
031	1F	037	071	47	107
032	20	040	072	48	110
033	21	041	073	49	111
034	22	042	074	4A	112
035	23	043	075	4B	113
036	24	044	076	4C	114
037	25	045	077	4D	115
038	26	046	078	4E	116
039	27	047	079	4F	117
040	28	050	080	50	120
041	29	051	081	51	121
042	2A	052	082	52	122
043	2B	053	083	53	123
044	2C	054	084	54	124
045	2D	055	085	55	125

Dezimal	Hex	Oktal	Dezimal	Hex	Oktal
086	56	126	126	7E	176
087	57	127	127	7F	177
088	58	130	128	80	200
089	59	131	129	81	201
090	5A	132	130	82	202
091	5B	133	131	83	203
092	5C	134	132	84	204
093	5D	135	133	85	205
094	5E	136	134	86	206
095	5F	137	135	87	207
096	60	140	136	88	210
097	61	141	137	89	211
098	62	142	138	8A	212
099	63	143	139	8B	213
100	64	144	140	8C	214
101	65	145	141	8D	215
102	66	146	142	8E	216
103	67	147	143	8F	217
104	68	150	144	90	220
105	69	151	145	91	221
106	6A	152	146	92	222
107	6B	153	147	93	223
108	6C	154	148	94	224
109	6D	155	149	95	225
110	6E	156	150	96	226
111	6F	157	151	97	227
112	70	160	152	98	230
113	71	161	153	99	231
114	72	162	154	9A	232
115	73	163	155	9B	233
116	74	164	156	9C	234
117	75	165	157	9D	235
118	76	166	158	9E	236
119	77	167	159	9F	237
120	78	170	160	A0	240
121	79	171	161	A1	241
122	7A	172	162	A2	242
123	7B	173	163	A3	243
124	7C	174	164	A4	244
125	7D	175	165	A5	245

Dezimal	Hex	Oktal	Dezimal	Hex	Oktal
166	A6	246	206	CE	316
167	A7	247	207	CF	317
168	A8	250	208	D0	320
169	A9	251	209	D1	321
170	AA	252	210	D2	322
171	AB	253	211	D3	323
172	AC	254	212	D4	324
173	AD	255	213	D5	325
174	AE	256	214	D6	326
175	AF	257	215	D7	327
176	B0	260	216	D8	330
177	B1	261	217	D9	331
178	B2	262	218	DA	332
179	B3	263	219	DB	333
180	B4	264	220	DC	334
181	B5	265	221	DD	335
182	B6	266	222	DE	336
183	B7	267	223	DF	337
184	B8	270	224	E0	340
185	B9	271	225	E1	341
186	BA	272	226	E2	342
187	BB	273	227	E3	343
188	BC	274	228	E4	344
189	BD	275	229	E5	345
190	BE	276	230	E6	346
191	BF	277	231	E7	347
192	C0	300	232	E8	350
193	C1	301	233	E9	351
194	C2	302	234	EA	352
195	C3	303	235	EB	353
196	C4	304	236	EC	354
197	C5	305	237	ED	355
198	C6	306	238	EE	356
199	C7	307	239	EF	357
200	C8	310	240	F0	360
201	C9	311	241	F1	361
202	CA	312	242	F2	362
203	CB	313	243	F3	363
204	CC	314	244	F4	364
205	CD	315	245	F5	365

Dezimal	Hex	Oktal	Dezimal	Hex	Oktal
246	F6	366	251	FB	373
247	F7	367	252	FC	374
248	F8	370	253	FD	375
249	F9	371	254	FE	376
250	FA	372	255	FF	377

Übungen

1. Fülle folgende Tabelle korrekt aus:

Dezimal	Hexadezimal	Oktal	Binär
	8FE7		
42391			
		13774	
			1101000110100110

2.4 Darstellung negativer Zahlen

Wir haben gesehen, wie vorzeichenlose ganze Zahlen durch Folgen von Binärziffern dargestellt werden. In einem Computer entspricht nun jeder Binärziffer ein *Bit*, Folgen von Bits werden meist *Bitketten* genannt. Mit einer Bitkette der Länge n (also bestehend aus den Bits b_0 bis b_{n-1}) können wir alle ganzen Zahlen im Bereich 0 bis 2^{n-1} darstellen.

Schwieriger ist es schon mit negativen Zahlen. Wir schreiben dabei ja ein *Minuszeichen* vor den Betrag der Zahl. Der Computer hat jedoch kein Minuszeichen, er hat nur Bitketten, in die er beliebige Inhalte füllen kann.

Die einfachste Möglichkeit ist, die menschliche Notation dadurch nachzuahmen, daß vor den Betrag der Zahl ein Bit gesetzt wird, das eine Codierung des Vorzeichens darstellt. Diese Form nennt man *Vorzeichen-Betrag-Darstellung*. Als Codierung für negatives Vorzeichen wird meist »1« verwendet, als Codierung für postives Vorzeichen »0«. Mit n Bits lassen sich dann Zahlen im Bereich $-(2^{n-1}-1)$ bis $+(2^{n-1}-1)$ darstellen. Die Null kommt dabei doppelt vor. Hier einige Beispiele:

```
 10 =                1010B
-10 =            10001010B mit n = 8
-10 = 1000000000001010B mit n = 16
```

$$
\begin{aligned}
127 &= \quad 1111111\text{B} \\
-127 &= \quad 11111111\text{B mit } n = 8 \\
-127 &= 1000000001111111\text{B mit } n = 16
\end{aligned}
$$

Die Vorzeichen-Betrag-Darstellung wird heute nur noch selten verwendet. Abgesehen von der Mehrdeutigkeit der Null, was den Test auf Vorliegen einer Null kompliziert, müssen bei arithmetischen Operationen stets mehrere Fälle unterschieden werden, was die Durchführung der Operationen verlangsamt und die zugehörigen Algorithmen beziehungsweise Programme unübersichtlich werden läßt. Die Vorzeichen-Betrag-Darstellung ist somit ein typisches Beispiel dafür, daß naheliegende Lösungen im Bereich der Programmierung nicht unbedingt die besten sind; Imitation menschlicher Vorgehensweise führt nicht notwendig auf gute Algorithmen für Computer.

Eine geschicktere Form der Darstellung ganzer Zahlen ist die sogenannte *1-Komplement-Darstellung.* Positive Zahlen werden dabei wie gewohnt dargestellt. Eine negative Zahl $-m$ wird bei Verwendung von n Bits durch 2^n-m-1 codiert. Der Übergang von m zu $-m$ geschieht einfach durch Invertieren der einzelnen Bits der Zahldarstellung (Invertieren bedeutet: aus »0« wird »1«, aus »1« wird »0«). Der darstellbare Zahlbereich geht von $-(2^{n-1}-1)$ bis $+(2^{n-1}-1)$. Die Null besitzt wieder zwei Darstellungen. Auch hier ist in Bit b_{n-1} das Vorzeichen der Zahl codiert. Allerdings stellt die Bitkette $b_{n-2}...b_1b_0$ *nicht* den Betrag der Zahl dar. Wir geben noch einige Beispiele für die 1-Komplement-Darstellung an:

$$
\begin{aligned}
10 &= \quad 1010\text{B} \\
-10 &= \quad 11110101\text{B mit } n = 8 \\
-10 &= 1111111111110101\text{B mit } n = 16
\end{aligned}
$$

$$
\begin{aligned}
127 &= \quad 1111111\text{B} \\
-127 &= \quad 10000000\text{B mit } n = 8 \\
-127 &= 1111111110000000\text{B mit } n = 16
\end{aligned}
$$

Die am häufigsten verwendete Darstellung ganzer Zahlen ist die *2-Komplement-Darstellung.* Auch hier werden positive Zahlen wie gewohnt dargestellt. Bei Verwendung von n Bits wird eine negative Zahl $-m$ durch 2^n-m codiert. Wir können also alle ganzen Zahlen z als durch z modulo 2^n dargestellt ansehen (modulo einer positiven Zahl k rechnen bedeutet, daß zu der zu reduzierenden Zahl so lange k beziehungsweise $-k$ addiert wird, bis das Ergebnis im Bereich 0 bis $k-1$ liegt; beispielsweise 23 modulo $4 = 3$, -17 modulo $8 = 7$, 12 modulo $3 = 0$). In dieser Form besitzt die Null nur noch eine Darstellung, dafür ist aber der Zahlbereich unsymmetrisch: er reicht von -2^{n-1} bis $+(2^{n-1}-1)$. An Bit b_{n-1} kann wieder das Vorzeichen abgelesen werden. Unter der Annahme, daß die Ergebnisse auszuführender Operationen wieder im 2-Komplement darstellbar sind, entsprechen Addition, Subtraktion und Multiplikation im 2-Komplement der ganz normalen Arithmetik des Binärsystems. Es braucht also nicht nach Vorzeichen unterschieden zu werden, man muß beim Ausführen der Operationen nicht einmal wissen, daß man ganze Zahlen statt vorzeichenloser ganzer Zahlen manipuliert (bei der Interpretation der

Zahlen natürlich schon!). Dies ist der große Vorteil der 2-Komplement-Darstellung gegenüber den anderen besprochenen Codierungen. Zum Abschluß noch einige Beispiele:

$$
\begin{aligned}
10 &= 1010B \\
-10 &= 11110110B \text{ mit } n = 8 \\
-10 &= 1111111111110110B \text{ mit } n = 16
\end{aligned}
$$

$$
\begin{aligned}
127 &= 1111111B \\
-127 &= 10000001B \text{ mit } n = 8 \\
-127 &= 1111111110000001B \text{ mit } n = 16
\end{aligned}
$$

Übungen

1. Versuche folgende vorzeichenlosen ganzen Zahlen binär mit einer Länge von 16 Bits darzustellen: 27233, 51896, 65983, 12356.

2. Gib folgende ganzen Zahlen mit einer Länge von 8 Bits in 2-Komplement-Darstellung, 1-Komplement-Darstellung und Vorzeichen-Betrag-Darstellung an (falls die jeweilige Darstellung existiert): 92, −123, 0, −128, 128.

3
Beschreibungsmittel

Vom Problem zum Programm ist es ein weiter Weg. Ausgehend von einer meist umgangssprachlichen Formulierung des Problems tastet man sich - insbesondere bei großen Programmsystemen - durch schrittweise Formalisierung an die Algorithmen heran, die dann zu einem Programm umgesetzt werden; letzteres verläuft meist ebenfalls in mehreren Phasen immer größeren Detaillierungsgrades.

Natürlich stellt eine umgangssprachliche Beschreibung eines Algorithmus keine besonders gute Ausgangsbasis für korrekte Programmierung dar. Man wird sich deshalb bemühen, einen Algorithmus immer möglichst schematisch zu beschreiben, und bei der Umsetzung zum Programm diese Formalisierung beizubehalten und zu vertiefen.

Für die formale Beschreibung von Algorithmen gibt es viele Möglichkeiten. Zwei davon sollen in diesem Buch benutzt werden: eine rein textuelle Beschreibungssprache (so etwas nennt man manchmal auch Pseudo-Code) und eine textuell/graphische Beschreibung in Form von Flußdiagrammen.

3.1 Eine formale Beschreibungssprache

Wir wollen jetzt eine formale Beschreibungssprache zur Formulierung von Algorithmen kennenlernen. Sie entspricht weitgehend den Konventionen der *strukturierten Programmierung* und lehnt sich stark an algorithmische Hochsprachen wie PASCAL an.

Wir beginnen mit unstrukturierten Anweisungen. Eine unstrukturierte Anweisung beschreibt im wesentlichen eine einzelne Aktion. Es können darin Register, Speicherzellen, Ports und Maschinenbefehle vorkommen, aber auch umgangssprachliche Umschreibungen von Vorgängen, die vielleicht in einem weiteren Schritt in mehrere Einzelaktionen aufgelöst werden. Ein Teil der Operationen erhält zwecks größerer Übersichtlichkeit formale Operationssymbole zugeordnet. Dies sind:

... <– ...	Transport des Ergebnisses des rechts vom Operator stehenden Ausdrucks in das links vom Operator stehende Ziel (Register, Speicherzelle oder Port; eventuell auch eine formale Variable).
< ... >	Inhalt eines Registers, einer Speicherzelle oder eines Ports (eventuell auch einer formalen Variablen).
(...)	Durch Speicheradresse bezeichnete Speicherzelle.
[...]	Durch Portadresse bezeichneter Port.
&	Konkatenation von Registern und Speicherzellen.
and	Bitweise Konjunktion (Logische Und-Verknüpfung).
or	Bitweise Disjunktion (Logische Oder-Verknüpfung).
xor	Bitweise exklusive Disjunktion (Logische Entweder-oder-Verknüpfung).
not	Bitweise Negation (Logische Nicht-Verknüpfung).

Dazu gleich einige Beispiele (zur Bedeutung der logischen Operatoren sei auf Kapitel 12.4 verwiesen):

A <– <A> + 2	Der Inhalt des A-Registers wird um 2 erhöht.
A <– <A> **and** 11011011B	Die bitweise Konjunktion des Inhalts des A-Registers mit der Maske 11011011B wird ins A-Register zurückgeschrieben.
Gewinn <– <Umsatz> – <Kosten>	Die Differenz der Inhalte zweier formaler Variablen »Umsatz« und Kosten« wird an die formale Variable »Gewinn« zugewiesen.
B & C & D & E <– <IX & IY>	Die 16-Bit-Register IX und IY werden zu einem 32-Bit-Register konkateniert (IX stellt dabei den höherwertigen Anteil dar); der Inhalt dieses *Superregisters* wird in ein anderes 32-Bit-Superregister – gebildet aus den 8-Bit Registern B, C, D, E – transportiert.
(7476H) <– <A>	Der Inhalt des A-Registers wird in die Speicherzelle mit der Adresse 7476H geschrieben.
A <– <(248AH)>	Der Inhalt der Speicherzelle mit der Adresse 248AH wird ins A-Register übertragen.
A <– <A> + <(<HL>)>	Der Inhalt des A-Registers wird um den Inhalt derjenigen Speicherzelle erhöht, deren Adresse im HL-Register steht.
(<HL>) <– 0	In die Speicherzelle, deren Adresse im HL-Register steht, wird eine Null geschrieben.

(<HL>) <– <(<HL>)> - 1	Der Inhalt der Speicherzelle, deren Adresse im HL-Register steht, wird um 1 vermindert.
[34H] <– <A>	Der Inhalt des A-Registers wird auf den Port mit der Portadresse 34H ausgegeben.
A <– <[02H]>	Der Inhalt des Ports mit der Portadresse 02H wird ins A-Register gebracht.
[<C>] <– <B>	Der Inhalt des B-Registers wird auf den Port ausgegeben, dessen Portadresse im C-Register steht.
H <– <[<C>]>	Der Inhalt des Ports, dessen Portadresse im C-Register steht, wird ins H-Register gebracht.

Mit einer unstrukturierten Anweisung kann man normalerweise keine vollständigen Algorithmen beschreiben. Deshalb setzt man mehrere Anweisungen zu strukturierten Anweisungen zusammen (die Teile einer strukturierten Anweisung können ebenfalls wieder strukturierte Anweisungen sein). Die einfachste Art, dies zu tun, ist die *Sequenz*, die durch linksbündiges Untereinanderschreiben der einzelnen Anweisungen notiert wird:

$Anweisung_1$
$Anweisung_2$
:
:
:
$Anweisung_n$

Die einzelnen Anweisungen einer Sequenz werden nacheinander abgearbeitet. Dazu ein Beispiel in umgangssprachlicher Formulierung: Wenn wir telefonieren, dann bedeutet das

Hörer abnehmen
Geld einwerfen
Nummer wählen
Gespräch führen
Hörer einhängen
Restgeld entnehmen

Hängt der weitere Verlauf einer Aktionsfolge vom Ergebnis einer vorausgegangenen Aktion ab, so benötigen wir eine *Verzweigung*. Eine *einseitige Verzweigung* liegt vor, wenn eine Anweisung, die natürlich auch eine strukturierte Anweisung sein kann, nur dann ausgeführt werden soll, wenn eine bestimmte Bedingung erfüllt ist:

wenn Bedingung
dann Anweisung

Manchmal möchten wir aber auch, daß bei nicht erfüllter Bedingung alternativ eine andere Anweisung ausgeführt wird. Dies ist dann eine *zweiseitige Verzweigung*:

wenn Bedingung
dann Anweisung$_1$
sonst Anweisung$_2$

Wir geben auch hier umgangssprachliche Beispiele, zuerst für eine einseitige Verzweigung: Wir kommen nach Hause, also

wenn Türe verschlossen
dann Türe aufschließen
Türe öffnen
Haus betreten

Allerdings kann die Situtation auch komplizierter sein, wenn wir jemand anders besuchen, was auf eine zweiseitige Verzweigung führt:

Klingeln
wenn Türe wird geöffnet
dann Haus betreten
sonst wieder weggehen

Es gibt auch Aktionen, die (mit oder ohne Variation) öfters wiederholt werden müssen. Solche Probleme modellieren wir durch *Schleifen.* Es gibt da zunächst die Form der *abweisenden Schleife*: Solange eine bestimmte Bedingung erfüllt ist, soll eine dazugehörige Anweisung wiederholt werden:

wiederhole
Anweisung
solange Bedingung

Dies bedeutet, daß die Anweisung unter Umständen gar nicht ausgeführt wird, wenn nämlich die Bedingung gleich zu Anfang nicht erfüllt ist.

Der geplagte Computer-Freak (ich zitiere aus meinem Leben!) bedient sich folgender Methode:

wiederhole
reklamiere beim Hersteller
schicke Computer ein
hole reparierten Computer ab
probiere Computer aus
solange Computer defekt

Eine zweite Form ist die *annehmende Schleife*, die stets mindestens einmal durchlaufen wird, die aber terminiert, sobald eine bestimmte Bedingung erfüllt ist:

wiederhole
    Anweisung
bis Bedingung

Auch dazu noch ein Beispiel aus dem täglichen Leben: In den öffentlichen Schwimmbädern ist man aus Sparsamkeit dazu übergegangen, daß die Duschen nur noch nach Drücken eines Knopfes Wasser von sich geben, und dann jeweils nur für wenige Sekunden. Der Algorithmus für Schwimmbadbenutzer heißt also

wiederhole
    drücke Knopf
    dusche fünf Sekunden
bis keine Lust mehr

Eine *Zählschleife* führt die Anweisung eine bestimmte Anzahl von Malen aus. Dies wird meist so organisiert, daß es einen Schleifenzähler gibt, der einen Startwert erhält, und der nach jedem Durchlauf um eine bestimmte Schrittweite erhöht wird. Überschreitet der Schleifenzähler dabei einen vorgegebenen Endwert, so terminiert die Schleife. Formal wird die Zählschleife folgendermaßen beschrieben:

wiederhole
    Anweisung
mit Schleifenzähler von Startwert **bis** Endwert in Schritten von Schrittweite

Wollen wir beispielsweise die Summe der ungeraden Zahlen zwischen 15 und 87 bestimmen, so schreiben wir:

Summe <– 0
wiederhole
    Summe <– <Summe> + <Zahl>
mit Zahl von 15 **bis** 87 **in Schritten von** 2

Genau besehen handelt es sich hierbei um eine *aufsteigende Zählschleife*, weil der Schleifenzähler mit jedem Schleifendurchlauf wächst. Genausogut könnten wir aber auch von oben herunterzählen. Wir benutzen dazu die gleiche Form von Zählschleife, allerdings mit einer negativen Schrittweite. Das Vorzeichen der Schrittweite entscheidet also, ob eine *aufsteigende* oder eine *absteigende Zählschleife* vorliegt.

Als letzte Form existiert noch die *endlose Schleife*, die bei periodischen Vorgängen benutzt wird. Sie terminiert niemals:

wiederhole
    Anweisung

Ein typisches Beispiel geben die an manchen Gebäuden angebrachten Digitaluhren mit eingebautem Thermometer ab:

wiederhole
    Zeige aktuelle Uhrzeit an
    Zeige aktuelle Temperatur an

Bei komplizierten Schleifen kann es vorkommen, daß während der Abarbeitung ein Ausnahmefall eintritt, und die Schleife dann sofort beendigt werden soll (zum Beispiel wenn eine zu invertierende Matrix sich als singulär erweist). Dies läßt sich durch folgende Anweisung realisieren:

verlasse Schleife

Bei der Lösung komplizierter Probleme zerlegt man das ursprüngliche Problem meist in einige kleinere Probleme, die getrennt gelöst werden können. Die Algorithmen, die zu den Teilproblemen gehören, setzt man dann in *Unterprogramme* um, die abgeschlossene Einheiten (im Prinzip eigene Programme) darstellen. Jedes Unterprogramm erhält einen *Namen* und eine *Liste von Parametern*, das sind formale Variablen, die Werte aus dem Hauptprogramm ins Unterprogramm und zurück transportieren. Formal:

Unterprogramm Unterprogrammname (Formalparameter$_1$,..., Formalparameter$_n$)

Das Ende eines Unterprogramms wird einfach durch die Anweisung

Ende Unterprogramm

gekennzeichnet. Auch Unterprogramme kann man wie Schleifen an einer beliebigen Stelle abbrechen mittels

Verlasse Unterprogramm

Der *Aufruf* eines Unterprogramms erfolgt unter Angabe seines Namens und der Werte, die für die Formalparameter eingesetzt werden sollen:

aktiviere Unterprogrammname (Aktualparameter$_1$,...,Aktualparameter$_n$)

Zum Beispiel wollen wir ein Unterprogramm schreiben, das die Multiplikation zweier ganzer Zahlen durchführt. Das sieht etwa so aus:

Unterprogramm MULT (X, Y, Z)
    Z <– <X> * <Y>
Ende Unterprogramm

Mögliche Aufrufe wären:

aktiviere MULT (15, 8, A)
aktiviere MULT (<B>, <C>, HL)
aktiviere MULT (<BC>, <DE>, DE & HL)

Bei all dem sollte man immer im Auge behalten, daß die exakteste Darstellung eines Algorithmus immer das fertige Programm selbst ist, die in formaler Notation geschriebenen Algorithmen damit möglicherweise »Unschärfen« enthalten, das heißt interpretationsbedürftige Teile.

3.2 Flußdiagramme

Vielen Leuten sind Beschreibungssprachen zu unanschaulich, sie wollen auf den ersten Blick Informationen über den Kontrollfluß eines Programms gewinnen. Außerdem gibt es Situationen, in denen die Beschreibungssprache aus Kapitel 3.1 das Problem nicht detailliert genug beschreibt; dies betrifft zum Beispiel den Einsprung in ein Programmstück oder Unterprogramm. Flußdiagramme erlauben die Behandlung auf einer solch tiefen Ebene der Programmierung. Als Preis dafür macht es mehr Mühe, ein Flußdiagramm zu entwerfen und zu zeichnen, es nimmt relativ viel Platz weg, und bei großen Programmen ist es gänzlich unübersichtlich. Flußdiagramme sollten deshalb durchweg nur bei »kleinen« Problemen benutzt werden (in diesem Buch gibt es, vielleicht mit Ausnahme einiger Beispiele aus Kapitel 29, nur »kleine« Probleme). Natürlich kann auch ein großes, gut strukturiertes Problem durch intensive Verwendung von Unterprogrammen (lokal) »klein« gehalten werden.

Nun wollen wir uns die einzelnen Beschreibungselemente eines Flußdiagramms ansehen: Eine einzelne Aktion wird durch ein kleines Rechteck dargestellt, in dessen Innerem die Aktion formal oder umgangssprachlich beschrieben ist. Das Beschreibungselement hat einen Eingang und einen Ausgang, die durch Pfeile angedeutet werden. Eine solche einzelne Aktion wäre zum Beispiel:

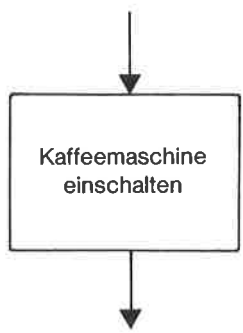


Bild 3.1. *Flußdiagramm: Beispiel einer einzelnen Aktion*

Wie schon in der Beschreibungssprache können einzelne Aktionen sequentiell ausgeführt werden. Dazu werden die Ausgänge der einzelnen Aktionen mit den Eingängen der jeweils nächsten Aktion verbunden. Start und Ende des Verfahrens werden durch ovale Beschreibungselemente markiert. Wir setzen unser Beispiel zu einer Sequenz fort:

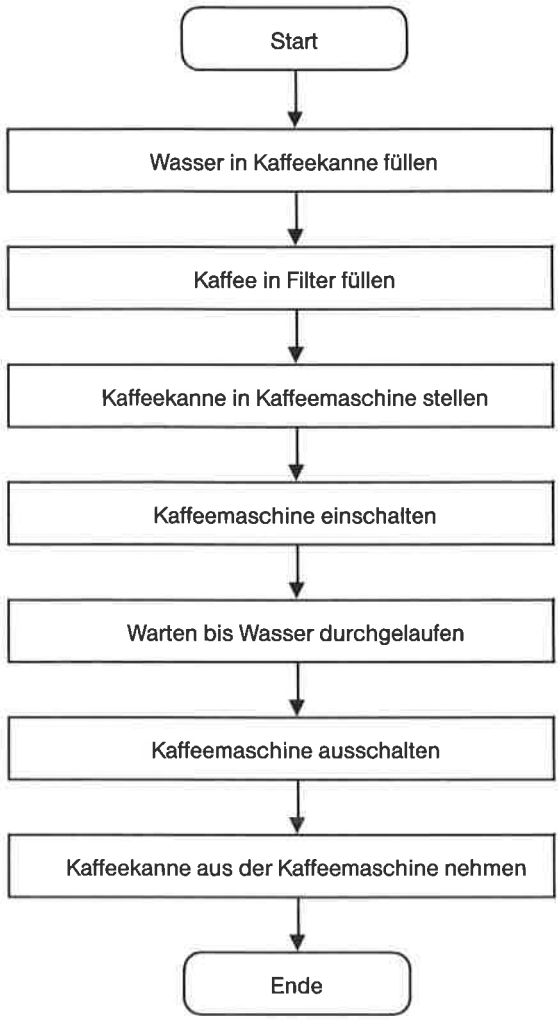


Bild 3.2. *Flußdiagramm: Beispiel einer Sequenz*

Eine weitere Aktionsmöglichkeit ist die Fallunterscheidung. Abhängig von einer Bedingung wird einer von zwei alternativen Wegen beschritten. Das Symbol für die Fallunterscheidung ist eine Raute. Diese hat demnach einen Eingang und zwei Ausgänge. Beispielsweise könnte in obiges Verfahren folgende Entscheidung eingebaut werden:

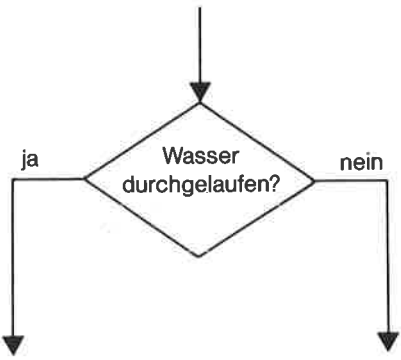


Bild 3.3. *Flußdiagramm: Beispiel einer Fallunterscheidung*

Die Ausgänge müssen nicht unbedingt zu beiden Seiten der Raute liegen; wichtig ist, daß sie durch »ja« und »nein« gekennzeichnet sind. Bei Verwendung von Fallunterscheidungen wird nicht mehr jeder Ausgang eines Beschreibungselements direkt mit einem Eingang eines anderen Beschreibungselements verbunden. Der Pfeil endet unter Umständen bereits an einem anderen Pfeil; diese Verbindungspunkte werden wir in unseren Assemblerprogrammen als Sprungziele wiederfinden.

Wir verfeinern nun das Element »Warten bis Wasser durchgelaufen« aus Bild 3.2 zu einer Schleife (erkennen Sie, welchen Typ sie darstellt?), unter Benutzung einer Fallunterscheidung:

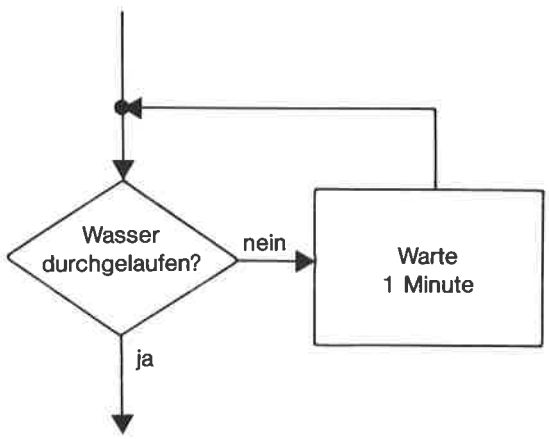


Bild 3.4. *Flußdiagramm: Beispiel einer einfachen Schleife*

Wir zeigen nun anhand des Schemas eines Flußdiagramms, daß nicht alle Flußdiagramme durch unsere formale Beschreibungssprache ausgedrückt werden können (ein typisches Phänomen für rückgekoppelte Verfahren!):

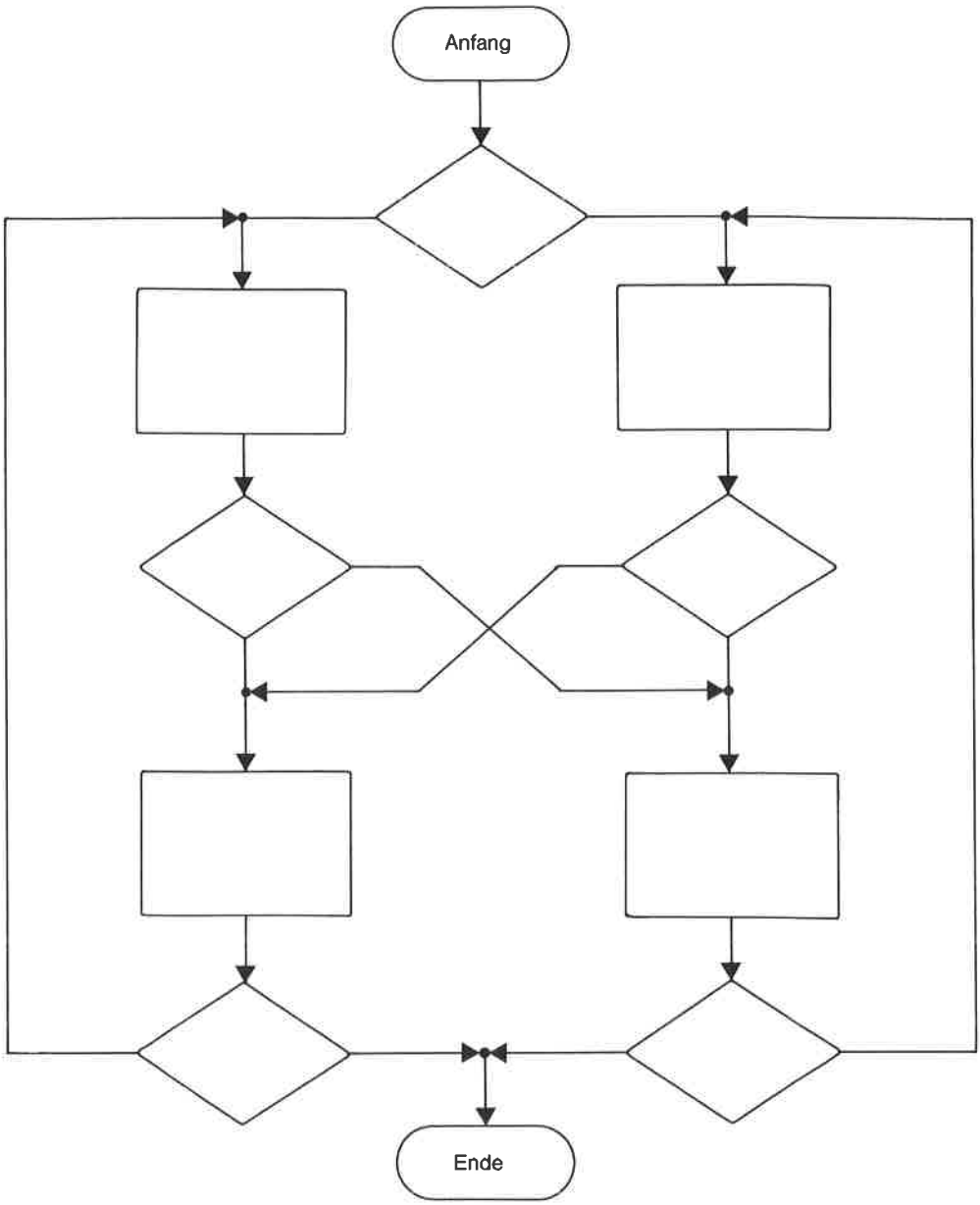


Bild 3.5. *Flußdiagramm: Schema eines rückgekoppelten Verfahrens*

Bei komplizierten Flußdiagrammen führt man Hilfspunkte ein, die mit Namen versehen werden. Diese Hilfspunkte werden durch kleine Kreise dargestellt. Jede Menge von solchen Kreisen mit demselben Namen muß die Eigenschaft besitzen, daß mindestens ein Pfeil an jedem dieser Kreise endet, aber genau aus einem der Kreise ein Pfeil (und zwar genau einer!) herausführt. Also beispielsweise:

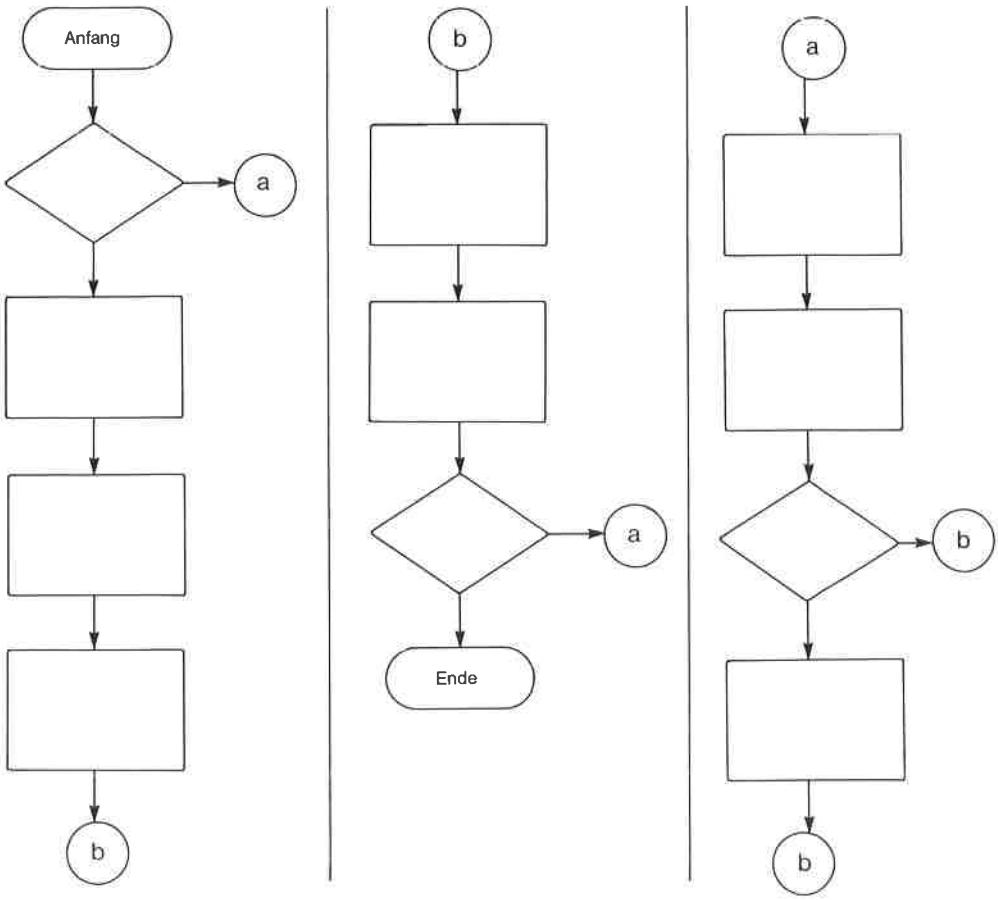


Bild 3.6. *Flußdiagramm: Beispiel für Hilfspunkte*

Die Einführung von Hilfspunkten gestattet es, große Flußdiagramme in mehrere Teile zu zerlegen und auf mehrere Seiten zu verteilen. Ein ähnlicher Effekt tritt ein, wenn wir das Problem durch Aufteilung in Teilprobleme in seiner Komplexität reduzieren; die einzelnen Teilprobleme werden durch Unterprogramme realisiert. Die Darstellung eines Unterprogramms sieht aus wie die eines Hauptprogramms; allerdings wird statt der Bezeichnung »Start« der Name des Unterprogramms verwendet. Der Aufruf eines Unterprogramms wird durch folgendes sechseckiges Beschreibungselement notiert:

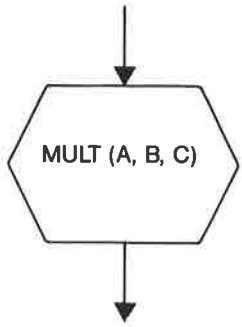


Bild 3.7. *Flußdiagramm: Aufruf eines Unterprogramms*

Nun zum Abschluß noch einige Feinheiten:

Kreuzen sich zwei Linien eines Flußdiagramms, so hat dies nur dann eine Bedeutung, wenn am Kreuzungspunkt mindestens ein Pfeil endet.

Für die Darstellung von Ein-/Ausgabe-Aktionen verwendet man statt des Rechtecks ein Trapez:

Bild 3.8. *Flußdiagramm: Beschreibungselement für Ein-/Ausgabe*

4
Der Prozessor Z80

In diesem Kapitel soll vom Prozessor Z80 beschrieben werden, was der Programmierer später von ihm sehen wird: Register und Befehlssatz. Dazu treten Assembler-Notation und die Form der Programm-Listings, die in diesem Buch durchgängig verwendet wird. Von Hardware-Gegebenheiten wird völlig abstrahiert. Wir lernen also das Programmieren, indem wir das Verhalten des Prozessors studieren (behavioristischer Ansatz, Black-box-Methode).

4.1 Register-Struktur des Z80

Der Z80 besitzt einen 8-Bit-Datenbus und einen 16-Bit-Adreßbus, was einen Adreßraum von 64 KB ergibt. Der Adreßraum ist nicht segmentierbar.

Es gibt folgende 8-Bit-Register:

A, F, B, C, D, E, H, L, A', F', B', C', D', E', H', L', I, R

Die Register können zum Teil paarweise zu Doppelregistern zusammengefaßt werden. Es gibt folgende Doppelregister:

AF (A und F), BC (B und C), DE (D und E), HL (H und L),
AF' (A' und F'), BC' (B' und C'), DE' (D' und E'), HL' (H' und L')

Mit den sekundären Registern A', F', B', C', D', E', H', L' kann nicht direkt gearbeitet werden; sie lassen sich nur durch bestimmte Befehle mit den entsprechenden Hauptregistern austauschen. Außer in den Austausch-Befehlen kommen die sekundären Register nicht als Argumente von Befehlen vor. Sie werden deshalb im folgenden (vorerst) nicht weiter behandelt.

Es gibt folgende 16-Bit-Register:

IX, IY, SP, PC

Zusätzlich besitzt der Z80 noch ein Unterbrechungs-Flipflop IFF (interrupt flipflop), dessen Zustand angibt, ob Unterbrechungen zugelassen sind.

Hier eine schematische Übersicht über die Register des Z80:

Hauptregister		Sekundäre Register	
Akkumulator A	Flags F	Akkumulator A'	Flags F'
B	C	B'	C'
D	E	D'	E'
H	L	H'	L'

Befehlszähler	PC
Stapelzeiger	SP
Index-Register	IX
Index-Register	IY

Unterbrechungs-vektor I
Speicher-Auffrisch-Register R

Bild 4.1. *Die Register des Z80*

Nur wenige der Register sind ausgesprochene Spezialregister. Dies sind:

PC	Befehlszähler (program counter)
SP	Stapelzeiger (stack pointer)
I	Unterbrechungs-Vektor (interrupt vector register)
R	Speicher-Auffrisch-Register (memory refresh register)
F	Flag-Register

Die übrigen Register sind multifunktional, jedoch nicht für alle Befehle benutzbar. Die ansprechbaren Funktionen sind im einzelnen:

A	Akkumulator für Arithmetik/Logik, Ein-/Ausgabe
B	Zählregister, Arithmetik/Logik, Ein-/Ausgabe
C	Portadreß-Register, Arithmetik/Logik, Ein-/Ausgabe
D	Arithmetik/Logik, Ein-/Ausgabe
E	Arithmetik/Logik, Ein-/Ausgabe
H	Arithmetik/Logik, Ein-/Ausgabe
L	Arithmetik/Logik, Ein-/Ausgabe
BC	Zählregister, Datenadreß-Register, 16-Bit-Arithmetik
DE	Datenadreß-Register, 16-Bit-Arithmetik
HL	Sprungadreß-Register, Datenadreß-Register, Akkumulator für 16-Bit-Arithmetik
IX	Index-Register, Sprungadreß-Register
IY	Index-Register, Sprungadreß-Register

Die beiden Flag-Register F und F' haben folgenden Aufbau:

Tabelle 4.1. *Aufbau der Flag-Register*

Bit	Flag	Bedeutung
7	S	Vorzeichen-Flag (sign flag)
6	Z	Null-Flag (zero flag)
5		nicht benutzt
4	H	Hilfs-Übertrag-Flag (half-carry flag)
3		nicht benutzt
2	P	Paritäts/Überlauf-Flag (parity/overflow flag)
1	N	Subtraktions-Flag (subtract flag)
0	C	Übertrag-Flag (carry flag)

Zur Unterscheidung von den Registern »C« beziehungsweise »H« wird in formaler Notation das Übertrag-Flag stets durch »CY« gekennzeichnet, das Hilfs-Übertrag-Flag durch »AC«.

4.2 Die Z80-Assembler-Notation von ZILOG

Im folgenden sollen die Konventionen des Standard Z80-Assemblers der Firma ZILOG beschrieben werden; die meisten anderen Z80-Assembler können Programme verarbeiten, die in dieser Notation verfaßt sind (umgekehrt meist nicht!).

Jedes Assemblerprogramm besteht aus einer Folge von Anweisungen (engl. statements).

Diese Anweisungen richten sich entweder an den Z80-Prozessor – sind also Codierungen von Maschinenbefehlen – oder an den Assembler selbst (sogenannte *Pseudo-Operationen*). Mittels der Pseudo-Operationen wird der Ablauf des Assemblierungsvorgangs gesteuert.

Jede Anweisung steht in einer separaten Zeile des Quell-Textes und ist in einzelne Felder gegliedert. Siehe dazu folgendes Beispiel eines Unterprogramms:

```
Marke     Opera-   Operanden     Kommentar
          tion
ANFANG:   LD       A,(WERT1)     ; ersten Operanden laden
          LD       B,A           ; in B sichern
          LD       A,(WERT2)     ; zweiten Operanden laden
          ADD      A,B           ; Summe bilden
          LD       (SUMME),A     ; Summe speichern
          RET                    ; Unterprogramm verlassen
WERT1:    DEFS     1             ; Speicherplatz
                                 ; fuer den ersten Operanden
WERT2:    DEFS     1             ; Speicherplatz
                                 ; fuer den zweiten Operanden
SUMME:    DEFS     1             ; Speicherplatz fuer die Summe
```

Das Operationsfeld enthält den Operationsnamen eines Maschinenbefehls oder einer Pseudo-Operation. Zwischen Operationsname und Operandenfeld muß mindestens ein Leerzeichen oder Tab stehen.

Das Operandenfeld enthält einen oder mehrere Operanden für die Operation (zum Beispiel Daten oder Adressen). Es kann auch leer sein, wenn nämlich die Operation keine Operanden benötigt. Die einzelnen Operanden werden durch Kommas voneinander getrennt.

Markenfeld und Kommentarfeld können ebenfalls leer sein. Das Markenfeld enthält entweder eine Marke oder (für die Pseudo-Operationen »EQU« und »DEFL«) einen symbolischen Namen.

Eine Marke ist ein frei wählbarer Name, gefolgt von einem Doppelpunkt. Sie stellt eine symbolische Bezeichnung für diejenige Speicheradresse dar, an welcher der markierte Befehl später im Objekt-Code zu liegen kommt, und dient anderen Anweisungen als Bezugsmöglichkeit auf eben diese Speicheradresse.

Namen bestehen aus bis zu sechs Zeichen. Das erste Zeichen muß ein Buchstabe sein; alle weiteren Zeichen können Buchstaben, Ziffern, das Fragezeichen oder der Unterstreichungsstrich sein. Sicherheitshalber sollte man große Buchstaben verwenden.

Einige Namen werden als Schlüsselwörter verwendet und dürfen daher nicht vom Programmierer benutzt werden. Dies sind die 8-Bit-Register-Namen (A, B, C, D, E, F, H, I, L, R), die 16-Bit-Register-Namen (IX, IY, PC, SP), die Doppelregister-Namen (AF, BC, DE, HL) und die Zustände der vier abfragbaren Flags C, Z, S und P (C, NC, Z, NZ, M, P, PE, PO).

Das Kommentarfeld enthält Erläuterungen des Programmierers zur entsprechenden Anweisung. Vor dem Kommentar muß zur Kennzeichnung ein Strichpunkt stehen.

Eine Anweisung kann auch alleine aus einem Kommentar bestehen, der zur vorhergehenden oder nachfolgenden Anweisung gehört.

Das obenstehende Programmbeispiel wurde mit Hilfe von Tabulatoren formatiert. Dies ist allgemein nicht erforderlich, hilft aber, das Programm lesbar zu machen.

4.3 Darstellung des Objekt-Codes

Wir werden manchmal zu den Quell-Programmen auch die daraus erzeugten Objekt-Programme angeben. Dabei gehen wir meist davon aus, daß der Objekt-Code ab der Adresse 0000H steht. Für andere Anfangsadressen müssen alle absoluten Adressen des Programms im Objekt-Code entsprechend angepaßt werden.

Pseudo-Operationen führen nicht unbedingt zur Erzeugung von Objekt-Code. Der Objekt-Code eines Maschinenbefehls ist zwischen ein und vier Bytes lang. Die spezielle Form der Listings entnehme man folgendem Beispiel aus Kapitel 8:

```
DREI:   LD      A,12        ; Operand initialisieren
        LD      D,A         ; Operand sichern
        ADD     A,A         ; Operand verdoppeln
        ADD     A,D         ; Operand verdreifachen
```

Der zugehörige Objekt-Code wird folgendermaßen dargestellt:

Adresse	Objekt-Code	Marke	Anweisung	
0000	3E 0C	DREI:	LD	A,12
0002	57		LD	D,A
0003	87		ADD	A,A
0004	82		ADD	A,D

Adressen und Objekt-Code werden stets in Hex angegeben (das angehängte »H« unterdrükken wir dabei).

4.4 Überblick über die Befehlsgruppen des Z80

Im folgenden soll der Befehlsvorrat des Z80 knapp umrissen werden. Einzelheiten zu den jeweiligen Befehlen werden im weiteren Verlauf des Buches beziehungsweise im Anhang erläutert.

Die Gruppe der 8-Bit-Lade-Befehle enthält folgende Klassen von Befehlen:

- Laden eines 8-Bit-Registers mit einem konstanten 8-Bit-Datenwert

- Laden einer Speicherzelle mit einem konstanten 8-Bit-Datenwert
- Transport des Inhalts eines 8-Bit-Registers in ein anderes 8-Bit-Register
- Transport des Inhalts eines 8-Bit-Registers in eine Speicherzelle
- Transport des Inhalts einer Speicherzelle in ein 8-Bit-Register

Die Gruppe der 16-Bit-Lade-Befehle enthält folgende Klassen von Befehlen:
- Laden eines 16-Bit-Registers oder Doppelregisters mit einem konstanten 16-Bit-Datenwert
- Transport des Inhalts eines 16-Bit-Registers oder Doppelregisters in den Stapelzeiger
- Transport des Inhalts eines 16-Bit-Registers oder Doppelregisters in zwei aufeinanderfolgende Speicherzellen
- Transport des Inhalts von zwei aufeinanderfolgenden Speicherzellen in ein 16-Bit-Register oder Doppelregister
- Transport des Inhalts eines 16-Bit-Registers oder Doppelregisters auf den Stapel
- Laden eines 16-Bit-Registers oder Doppelregisters aus dem Stapel

Die Gruppe der Austausch-Befehle enthält folgende Klassen von Befehlen:
- Austausch des Inhalts eines 16-Bit-Registers oder Doppelregisters mit einem Stapelelement
- Austausch der Inhalte von Hauptregistern und sekundären Registern
- Austausch der Inhalte von Doppelregistern

Die Gruppe der Befehle für blockweises Bewegen dient dem Transport eines zusammenhängenden Speicherbereichs mit Hilfe eines einzigen Maschinenbefehls.

Die Gruppe der Such-Befehle erlaubt das Absuchen eines zusammenhängenden Speicherbereichs nach einem bestimmten 8-Bit-Datenwert.

Die Gruppe der 8-Bit-Arithmetik- und Logik-Befehle enthält folgende Klassen von Befehlen:
- Addition eines konstanten 8-Bit-Datenwerts, Inhalts eines 8-Bit-Registers oder Inhalts einer Speicherzelle zum A-Register
- Subtraktion eines konstanten 8-Bit-Datenwerts, Inhalts eines 8-Bit-Registers oder Inhalts einer Speicherzelle vom A-Register
- Vergleich eines konstanten 8-Bit-Datenwerts, Inhalts eines 8-Bit-Registers oder Inhalts einer Speicherzelle mit dem Inhalt des A-Registers
- Logische Verknüpfung eines konstanten 8-Bit-Datenwerts, Inhalts eines 8-Bit-Registers oder Inhalts einer Speicherzelle mit dem A-Register
- Erhöhen des Inhalts eines 8-Bit-Registers oder einer Speicherzelle um 1
- Erniedrigen des Inhalts eines 8-Bit-Registers oder einer Speicherzelle um 1
- Negation des A-Registers im 1-Komplement oder 2-Komplement
- Korrektur des Inhalts des A-Registers entsprechend einer arithmetischen Operation für dezimal codierte ganze Zahlen
- Setzen oder Löschen des Übertrag-Flags

Die Gruppe der 16-Bit-Arithmetik und Logik-Befehle enthält folgende Klassen von Befehlen:
- Addition des Inhalts eines 16-Bit-Registers oder Doppelregisters zu einem anderen 16-Bit-Register oder Doppelregister

- Subtraktion des Inhalts eines 16-Bit-Registers oder Doppelregisters vom HL-Register
- Erhöhen des Inhalts eines 16-Bit-Registers oder Doppelregisters um 1
- Erniedrigen des Inhalts eines 16-Bit-Registers oder Doppelregisters um 1

Die Gruppe der Rotations- und Verschiebe-Befehle enthält folgende Klassen von Befehlen:
- Rotieren oder Verschieben des Inhalts eines 8-Bit-Registers
- Rotieren oder Verschieben des Inhalts einer Speicherzelle
- Austausch von 4-Bit-Größen zwischen A-Register und einer Speicherzelle

Die Gruppe der Bit-Manipulations-Befehle enthält folgende Klassen von Befehlen:
- Setzen, Löschen oder Testen eines Bits eines 8-Bit-Registers
- Setzen, Löschen oder Testen eines Bits einer Speicherzelle

Die Gruppe der Sprung-Befehle enthält folgende Klassen von Befehlen:
- Bedingte oder unbedingte absolute Sprünge zu einer fest vorgegebenen Programmadresse
- Absolute Sprünge zu einer Adresse, die im HL-Register oder einem Index-Register steht
- Bedingte oder unbedingte relative Sprünge mit einer fest vorgegebenen Sprungdistanz
- Schleifen-Befehl

Die Gruppe der Unterprogramm-Befehle enthält folgende Klassen von Befehlen:
- Bedingte oder unbedingte Unterprogramm-Aufrufe
- Bedingte oder unbedingte Unterprogramm-Rücksprünge
- Rücksprünge aus einer Unterbrechungs-Routine

Die Gruppe der Kontroll-Befehle enthält folgende Klassen von Befehlen:
- Unterbrechungs-Steuerung
- Anhalten des Prozessors
- Leer-Befehl

Die Gruppe der Ein-/Ausgabe-Befehle enthält folgende Klassen von Befehlen:
- Einlesen eines 8-Bit-Datenwerts aus einem Port in ein 8-Bit-Register
- Ausgeben des Inhalts eines 8-Bit-Registers auf einen Port
- Einlesen einer Folge von 8-Bit-Datenwerten aus einem Port in einen zusammenhängenden Speicherbereich
- Ausgeben des Inhalts eines zusammenhängenden Speicherbereichs auf einen Port

Der Befehlssatz des Z80 ist teilweise recht wenig regulär, das heißt zum Beispiel, daß es Befehle für bestimmte Doppelregister gibt, die für andere Doppelregister nicht bestehen. Vergleiche hierzu Anhang A!

4.5 Adressierungsarten

Der Z80 verfügt über eine Fülle von Adressierungsarten:

Zunächst einmal können alle Operanden in Registern stehen (Adressierungsart: register direct). Beispiele:

```
- LD      C,H
  SUB     E
- EXX
- SCF
- INC     BC
- DAA
- JP      (HL)
```

Als nächstes kann ein Operand selbst (8-Bit-Datenwert, 16-Bit-Datenwert, relative oder absolute Adresse) direkt auf den Operationscode folgen (Adressierungsart: immediate). Beispiele:

```
- LD      A,12
- LD      IY,35E8H
- SUB     15
- XOR     11001010B
- JP      2231H
- JR      24H
```

Es wird von anderen Autoren gelegentlich bestritten, daß ein absoluter oder relativer Sprung mit festem Sprungziel beziehungsweise fester Sprungdistanz unter die Adressierungsart »immediate« fällt. Bei genauem Hinsehen ist ein absoluter Sprung nichts anderes als ein Lade-Befehl für das Register PC, ein relativer Sprung dagegen eine arithmetische Operation auf dem Register PC. Wer's nicht glaubt, soll sich zum Beispiel den Assembler der PDP-11 ansehen!

Es kann aber auch ein Operand (8-Bit-Datenwert, 16-Bit-Datenwert) im Speicher stehen und durch seine Adresse spezifiziert werden, wobei diese wieder unmittelbar auf den Operationscode folgt (Adressierungsart: direct). Beispiele:

```
- LD      A,(8674H)
- LD      (1FFH),SP
```

Ebenfalls direkte Adressierung liegt vor, wenn auf den Operationscode eine Port-Adresse folgt. Beispiele:

```
- IN      A,(04H)
- OUT     (02H),A
```

Wird die Adresse (oder Port-Adresse) nicht im Befehl selbst mitgeliefert, sondern steht diese in einem Register, so spricht man von indirekter Adressierung (Adressierungsart: register indirect). Beispiele:

- LD A,(BC)
- DEC (HL)
- BIT 3,(HL)
- IN B,(C)
- OUT (C),L
- RLD

Es gibt einige Spezialbefehle, die implizit Gebrauch machen von der indirekten Adressierung mittels Register. Dort werden teilweise sogar zwei Doppelregister zur Adressierung von zwei Operanden verwendet und zusätzlich noch die Ausführung der Operation über weitere Register gesteuert. Beispiele:

- CPIR
- LDDR
- OTIR

Einige Operationen bringen Daten aus Doppelregistern auf den Stapel und umgekehrt. Hierbei wird über den Stapel-Zeiger indirekt adressiert. Beispiele:

- PUSH DE
- POP AF
- EX (SP),HL
- RET

Dies kann auch noch mit der Adressierungsart immediate verbunden sein. Beispiel:

- CALL 2732H

Bei der Adressierung über Index-Register (Adressierungsart: indexed) wird hinter dem Operationscode eine Relativadresse im Bereich -80H bis +7FH mitgeführt. Die Adresse des Operanden ergibt sich als Summe der im Index-Register stehenden Adresse und der Relativadresse. Beispiele:

- AND (IX+24H)
- LD (IY-3),L
- SET 4,(IX+8)
- SLA (IY+7EH)

In einigen Lade-Befehlen kann die Adressierungsart immediate mit einer der Adressierungsarten register indirect oder indexed gekoppelt werden. Beispiele:

- `LD (HL),0AH`
- `LD (IX+2),44`

5
Erstellen und Testen von Programmen

Im folgenden soll kurz beschrieben werden, welche Phasen beim Entwickeln, Erstellen und Testen eines Assemblerprogramms zu durchlaufen sind und welche Hilfsmittel dazu zur Verfügung stehen. Es schließt sich ein Abschnitt für diejenigen an, die nicht über die notwendigen Werkzeuge der Programmentwicklung verfügen, die aber trotzdem kleine Assembler-Routinen schreiben und implementieren wollen, zum Beispiel um in einem BASIC-Programm zeitkritische Funktionen schneller zu gestalten.

5.1 Methoden des Programmentwurfs

Die Methoden des Programmentwurfs haben sich im Verlauf der letzten zwanzig Jahre zu einer eigenständigen Wissenschaft, im Englischen »Software Engineering« genannt, entwickelt. »Große« Programme – und in der Praxis sind fast alle Programme groß – lassen sich nur durch Systematik in den Griff bekommen. Ich möchte deshalb hier einige Techniken vorstellen, die auch für kleinere Probleme mit Gewinn einsetzbar sind.

Was man zu Beginn eines Programmierprojekts vorliegen hat, ist meist nur eine mehr oder weniger unvollständige umgangssprachliche – oder auch nur gedankliche – Formulierung des Problems. Diese muß zunächst in eine formale Anforderungsdefinition (Spezifikation) umgesetzt werden; die Anforderungsspezifikation wird manchmal auch »Pflichtenheft« genannt.

Damit man aus der Spezifikation die Struktur des gesamten Problems ersehen kann, läßt man zunächst alle Details weg und beschränkt sich auf prinzipielle Vorgaben. Diese erste Spezifikation erweist sich beim weiteren Fortgang des Projekts als zu ungenau, eben weil die Details weggelassen wurden. Man füllt nun einige oder alle Teile der Spezifikation genauer aus und treibt so das Projekt einen Schritt voran. Dieser Verfeinerungsprozeß wird so lange wiederholt, bis in einer endgültigen Spezifikation alle Details genau beschrieben sind.

Bei der Umsetzung der Spezifikation in ein Programm geht man ähnlich vor. Beginnend mit

der gröbsten Anforderungsdefiniton werden Algorithmen entwickelt, in denen bestimmte Details noch nicht festgelegt sind. Werden die Algorithmen in Programme umgesetzt, so entsprechen den nicht ausgeführten Details sogenannte »Dummies«, das sind Programmstücke, die im Ein-/Ausgabe-Verhalten in etwa die Funktionen simulieren, die später an dieser Stelle notwendig sind. Wieder läuft der Entwicklungsprozeß in Phasen ab, während derer die Dummies nach und nach verschwinden und durch den endgültigen Code ersetzt werden.

Diese Vorgehensweise nennt man im Englischen *Top-Down-Design* (Entwicklung von oben nach unten, das heißt vom Allgemeinen zum Speziellen).

Unterstützt wird die schrittweise Verfeinerung der Spezifikation durch halb-formale Beschreibungssprachen, die einen formalen Rahmen vorgeben, in den entweder exakte Anweisungen oder umgangssprachliche Erläuterungen eingesetzt werden können, welche die exakten Anweisungen kommentieren oder ergänzen. Die Beschreibungssprache aus Unterkapitel 3.1 ist dazu geeignet.

Es hat sich als günstig herausgestellt, große Programme in kleinere Einheiten, sogenannte *Module*, zu zerlegen, die später zusammen ein Programm bilden (Modularisierung). Zwischen den Modulen bestehen externe Bezüge auf Variable, über welche die Module Daten untereinander austauschen können.

Diese Vorgehensweise wird fortgesetzt durch Zerlegen eines Moduls in *abgeschlossene Unterprogramme*, das sind Programmstücke mit einer bestimmten Aufgabe, die bei der Aktivierung Daten vom aufrufenden Haupt- oder Unterprogramm erhalten und nach Beendigung ihres Auftrags Ergebnisse an den Aufrufer zurückliefern.

Die Dummies bestehen oft aus einem Unterprogrammrahmen, in den hauptsächlich Kommentare und Befehle zur Simulation der Ergebnisse eingebaut sind.

Module und abgeschlossene Unterprogramme können getrennt voneinander erstellt und getestet werden. Dies hat den Vorteil, daß Fehler im Programm meist frühzeitig entdeckt werden und leicht lokalisierbar sind.

Techniken, mit denen die korrekte Übersetzung der Spezifikation in Programme überprüft werden kann, nennt man »Verifikations-Methoden«. Dieser Teilbereich der Programmentwicklung steckt wissenschaftlich noch in den Kinderschuhen. Ob die Spezifikation korrekt ist, kann allerdings nicht durch formale Methoden festgestellt werden. Ein verifiziertes Programm leistet damit noch nicht unbedingt das, was der Programmierer von ihm erwartet.

Es gibt einige Aufgaben, die in einem Programmsystem immer wieder anfallen, zum Beispiel Ein-/Ausgabe, Speicherverwaltung, Berechnung mathematischer Funktionen oder Fehlerbehandlung. Für diese Aufgaben schreibt man Unterprogramme, die in *Programm-Bibliotheken* gesammelt werden; diese Unterprogramme können nach Bedarf aus der jeweiligen Bibliothek kopiert und in ein Programm eingebaut werden. Besonders wichtig bei diesen Unterprogrammen ist eine ausführliche Dokumentation über Eingabedaten, Ergebnisse und Funktionsweise des Unterprogramms.

5.2 Werkzeuge der Programmentwicklung

Um ein Assemblerprogramm zu erstellen, brauchen wir zunächst einen *Editor*. Bei der Wahl eines Editors sollte man berücksichtigen, daß ein Text-Editor meist auch zur Programmentwicklung geeignet ist, ein Programm-Editor aber nur selten auch zur Erstellung anderer Texte taugt. Wer allerdings mit seinem Editor ausschließlich Programmentwicklung treiben möchte, dem sei geraten: ein einfacher Zeilen-Editor tut's zur Not auch!

Haben wir unser Programm mit dem Editor erstellt, so benötigen wir als nächstes einen *Assembler* (manche Assembler enthalten sogar einen Editor; diese heißen dann meist *Editor-Assembler*). Der Assembler prüft das Programm auf verschiedene Fehler wie unzulässige Operationsnamen, nicht definierte Marken oder Namen, inkompatible Operanden, falsche Zahl oder Typen von Operanden. Die Fehlermeldungen sind stets mit der Nummer der Zeile versehen, in welcher der Fehler entdeckt wurde (meistens steckt er dann auch in dieser oder der vorhergehenden Zeile).

Logische Fehler wie falsche Register oder vertauschte Reihenfolge von Anweisungen kann der Assembler allerdings nicht entdecken; diese Aufgabe ist so kompliziert, daß selbst auf Supercomputern praktisch keine derartigen Systeme existieren. Die häufigsten Fehler des Programmieranfängers sind jedoch Schreibfehler oder Mißachtung der Konventionen, und diese Fehler entdeckt der Assembler fast immer.

Ein zweiter Schritt ist dann die sogenannte Code-Erzeugung, also das Übersetzen der einzelnen Anweisungen in die korrespondierenden Maschinenbefehle. Wurde im Programm eine feste Anfangsadresse angegeben, so fertigt der Assembler ein fixiertes Objekt-Programm an, das nun beliebig oft geladen und gestartet werden kann. Wurde keine Anfangsadresse angegeben, so ist die Fixierung der Adressen die Aufgabe des Relokators. Der Assembler bereitet dann das Objekt-Programm soweit vor, daß nur noch an bestimmten Stellen die korrekten Speicheradressen eingesetzt werden müssen.

Es folgt das Binden des Programms durch den sogenannten *Binder*. Wie wir gesehen haben, ist Modularisierung eine wichtige und für viele Projekte sogar unentbehrliche Methode der Programmentwicklung. Bei modularer Programmierung liefert uns der Assembler den Zwischencode des Programms, zerlegt in Teile (die Module). Das Zusammenfügen der Module, die für ein Programm benötigt werden, ist meist mit einem Durchsuchen von *Bibliotheken* verbunden, in denen sich der Zwischencode von vorübersetzten Modulen für häufig durchzuführende Aufgaben wie Ein-/Ausgabe oder Berechnung mathematischer Funktionen befindet (eine Bibliothek könnte man in Anlehnung an den Begriff »Datenbank« anschaulich als »Programmbank« bezeichnen). Das Binden eines Programms kann ähnlich aufwendig sein wie das Assemblieren selbst; es werden dabei alle externen Bezüge der einzelnen Module korrekt eingesetzt (engl. external resolving).

Der *Relokator* rechnet nun mit Hilfe einer entweder von ihm selbst, vom Betriebssystem oder vom Benutzer vorgegebenen Anfangsadresse alle relativen Adressen (bezogen auf die Anfangsadresse 0000H) in die tatsächlichen, absoluten Adressen um, und setzt diese an den entsprechenden Stellen des Objekt-Codes ein. Das Objekt-Programm ist nun fixiert (gebunden). Der Relokator ist meist in den Binder oder den Lader integriert.

Durch einen *Lader* wird das Objekt-Programm an die richtige Stelle des Speichers gebracht,

wo es nun auf seine Ausführung wartet. Manchmal sind Binder und Lader zu einem *Binder-Lader* zusammengefaßt.

Ist das geladene Objekt-Programm ein Hauptprogramm, so kann es durch das Betriebssystem ausgeführt werden; ist es dagegen ein Unterprogramm, zum Beispiel für ein BASIC-Programm, so wird es zu gegebener Zeit durch Aufruf aus dem Hauptprogramm aktiviert und gibt nach Beendigung seines Auftrags die Kontrolle an dieses zurück.

Alle Programme (auch ganz kurze!) sollte man mit geeigneten Daten austesten. Das Finden realistischer Daten ist meist gar nicht einfach, eine allgemeine Strategie dazu wurde noch nicht gefunden. Das Hilfsmittel zum Testen ist der *Debugger* oder *Monitor*. Mit dem Debugger bringen wir unsere Testdaten in Speicherzellen und Register, überwachen den Ablauf kritischer Stellen im Programm und sehen uns hinterher die Ergebnisse an, die der Lauf produziert hat. Spätestens wenn ein im Einsatz befindliches Programm ein fehlerhaftes Verhalten zeigt, ist es an der Zeit, den Debugger zu benutzen.

Wir werden zu manchen Beispielen und Übungen Testdaten angeben und – wo immer möglich – erklären, warum diese geeignet sind (dies bedeutet natürlich nicht, daß man mit Hilfe dieser Testdaten alle Fehler eines Programms erkennen könnte).

5.3 Manuelle Assemblierung

Editor, Assembler, Binder, Lader und Debugger sind heutzutage meist recht billige Programme (das war nicht immer so). Trotzdem kann es sein, daß jemand nur ganz kleine Assemblerprogramme als Unterprogramme irgendwo einbauen oder vielleicht auch nur fehlerhaft gelieferte Software modifizieren möchte; er schreckt dann möglicherweise vor einer unnötigen Anschaffung zurück. Vielleicht ist aber auch gerade für seinen Computer kein Assembler auf dem Markt (zum Beispiel bei selbstgebauten Systemen). In diesem Fall kann man sich mit der – zugegebenermaßen etwas mühsamen – manuellen Assemblierung behelfen.

Bei der manuellen Assemblierung spielt der Programmierer selbst den Assembler. Wir wollen dies nun anhand eines Beispiels aus Kapitel 9 einmal vorführen (wenn Sie das Beispiel jetzt noch nicht ganz verstehen, macht das nichts).

Wir wollen annehmen, daß im A-Register ein ASCII-Zeichen steht (zu ASCII vergleiche Kapitel 7). Falls dieses kein Leerzeichen darstellt, soll es durch einen Punkt ersetzt werden. Das Unterprogramm zur Lösung dieses Problems könnte nun folgendermaßen lauten:

```
ERSETZ:   CP      20H      ; auf Leerzeichen pruefen
          JP      Z,LEER   ; Leerzeichen im A-Register
          LD      A,2EH    ; kein Leerzeichen,
                           ; durch Punkt ersetzen
LEER:     RET              ; Ende des Unterprogramms
```

Wir gehen nun ganz schematisch vor: Als erstes legen wir die Anfangsadresse fest (bei den meisten Programmen wird dies nicht die Adresse 0000H sein). Nehmen wir zum Beispiel die Anfangsadresse 4D00H. Wir schreiben diese in unserer Tabelle vor die erste Anweisung. Dann

sehen wir im Anhang B nach, wie viele Bytes Speicherplatz der Objekt-Code des ersten Befehls braucht (hier sind es 2 Bytes). Also kommt vor die nächste Anweisung die Adresse 4D02H. Der zweite Befehl benötigt 3 Bytes Speicherplatz, also steht vor der dritten Anweisung die Adresse 4D05H. So geht es weiter, bis vor allen Anweisungen die richtige Adresse steht:

```
Adresse   Objekt-Code   Marke      Anweisung
4D00                    ERSETZ:    CP      20H
4D02                               JP      Z,LEER
4D05                               LD      A,2EH
4D07                    LEER:      RET
```

Wenn wir das geschafft haben, so kennen wir schon die Werte aller Marken des Programms, können also alle absoluten Bezüge auf eine Marke durch die entsprechend Adresse ersetzen. Wir sehen nun den Objekt-Code der einzelnen Befehle im Anhang B nach; teilweise müssen wir den Objekt-Code mit Informationen ergänzen, die wir den Operanden des Befehls entnehmen, zum Beispiel die Werte 20H und 2EH sowie die Adresse LEER (in diesem Falle 4D07H). Damit ist das Programm auch schon assembliert:

```
Adresse   Objekt-Code   Marke      Anweisung
4D00      FE 20         ERSETZ:    CP      20H
4D02      CA 07 4D                 JP      Z,LEER
4D05      3E 2E                    LD      A,2EH
4D07      C9            LEER:      RET
```

Beim Ersetzen von 16-Bit-Größen beachten wir, daß die niederwertigen beiden Hex-Ziffern auch in das niederwertige Byte (LSB) kommen; nur so kann der Z80 mit 16-Bit-Größen richtig hantieren (siehe auch das Kapitel »Worte«). Prinzipiell wäre die Reihenfolge der Bytes allerdings beliebig.

Schwierigkeiten machen nun nur noch die relativen Sprünge und der Schleifen-Befehl. Wir müssen dazu die Relativadresse (Sprungdistanz), bezogen auf den nachfolgenden Befehl, berechnen und den Wert, als 2-Komplement dargestellt, im Objekt-Code eintragen (durch das Abarbeiten des Befehls wird der Befehlszähler um zwei erhöht). Die Sprungdistanz ist dabei die Adresse des auf den Sprungbefehl folgenden Befehls minus die Adresse des Sprungziels. Die Sprungdistanz kann damit auch negativ sein; sie muß aber stets im Bereich −128 bis +127 sein, sonst liegt ein Programmierfehler vor (die Relativadresse wird in einem Byte des Objekt-Codes abgespeichert). Ebenso müssen die Relativadressen von Index-Register-Befehlen im Bereich −128 bis +127 liegen.

Nun muß der Objekt-Code geladen werden. Nehmen wir an, daß unser Unterprogramm an ein BASIC-Programm angebunden werden soll, so ist es zweckmäßig, den Objekt-Code durch dieses Programm in den Speicher schreiben zu lassen. Dazu übersetzen wir erst einmal alle Hex-Zahlen in Dezimalzahlen. Das BASIC-Programm würde dann etwa so aussehen:

```
RESTORE
READ IA,IE
FOR I=IA TO IE
READ K
POKE I,K
NEXT I
DATA 19712,19719
DATA 254,32,202,7,77,62,46,201
END
```

Für ein anderes Assemblerprogramm müssen nur die DATA-Zeilen ausgetauscht werden.

Wie ein Unterprogramm aktiviert und mit Parametern versorgt wird, hängt von dem jeweiligen BASIC-Interpreter ab; versuche dies aus dem Handbuch zu erfahren (mögliche Stichworte: USR, DEFUSR, PEEK, POKE, CALL, SYS).

6
Bytes

Die kleinste Speichereinheit, die der Z80 mit einem Befehl zwischen internem und externem Speicher transportieren kann, ist ein Byte. Man bezeichnet den Prozessor deshalb auch als Byte-orientiert.

6.1 Erste Schritte: Der LD-Befehl

Mit Hilfe des Assemblerbefehls **LD** (load) können wir ein Byte in eines der 8-Bit-Register des Z80 bringen. Wollen wir zum Beispiel den Wert 12H in das A-Register laden, so lautet der entsprechende Befehl:

```
LD          A,12H       ; den Wert 12H
                        ; ins A-Register bringen
```

Der LD-Befehl hat stets zwei Argumente. Das erste Argument gibt an, wohin der Datenwert geschrieben werden soll (hier: das A-Register). Das zweite Argument spezifiziert den Datenwert selbst. In der Beschreibungssprache würde unser Beispiel lauten:

A <– 12H

Wir assemblieren nun dieses »Mini-Programm«:

Adresse	Objekt-Code	Marke	Anweisung
0000	3E 12		LD A,12H

Wer einen Debugger mit Einzelschrittausführung besitzt, sollte sich das Beispiel auch damit ansehen.

Noch ein Beispiel: der Datenwert C3H soll ins D-Register geschrieben werden. Der entsprechende Befehl lautet:

```
LD          D,0C3H      ; den Wert C3H
                        ; ins D-Register bringen
```

Beachte, daß Hex-Zahlen für den Assembler stets mit einer Ziffer beginnen, damit sie von Namen unterscheidbar sind!

Die numerischen Argumente von Befehlen können auch in dezimaler, binärer oder oktaler Notation angegeben werden. Wir könnten also statt obigen Befehls auch einen der folgenden drei Befehle schreiben (aus denen der Assembler stets denselben Objekt-Code erzeugt):

```
LD          D,195       ; den Wert C3H als Dezimalzahl
                        ; ins D-Register bringen
LD          D,11000011B ; den Wert C3H als Binaerzahl
                        ; ins D-Register bringen
LD          D,303Q      ; den Wert C3H als Oktalzahl
                        ; ins D-Register bringen
```

Merke: Durch einen LD-Befehl wird kein Flag verändert!

Übungen

1. Schreibe ein Programm, das den Wert 74H ins B-Register bringt.

2. Schreibe ein Programm, das den Wert F7H ins H-Register bringt.

3. Warum ist folgendes Programm sinnlos?

```
LD      16,B
```

4. Setze folgende Anweisung in ein Programm um:

   C <– D4H

5. Schreibe für folgende drei Anweisungen je ein Programm:

   E <– 96
   A <– 219Q
   E <– 1101011B

6.2 Einfache Byte-Arithmetik

Ein Datenwert vom Typ »Byte« kann als Darstellung einer vorzeichenlosen ganzen Zahl im Bereich 0 bis einschließlich 255 interpretiert werden (siehe Kapitel 2). Der Z80 verfügt deswegen über arithmetische Befehle, mit deren Hilfe man mit Werten vom Typ »Byte« rechnen kann. Die Berechnungen erfolgen stets modulo 256, damit das Ergebnis der Operation wieder vom Typ »Byte« ist.

Als erstes wollen wir die Addition kennenlernen. Mit Hilfe des Befehls **ADD** (add) können wir eine Konstante vom Typ »Byte« zum Inhalt des A-Registers addieren. Das Ergebnis wird wieder im A-Register abgelegt. Wollen wir zum Beispiel den Inhalt des A-Registers um 63H erhöhen, formal also

A <- <A> + 63H

so schreiben wir:

```
        ADD         A,63H       ; Inhalt des A-Registers
                                ; um den Wert 63H erhoehen
```

Der Objekt-Code sieht folgendermaßen aus:

Adresse	Objekt-Code	Marke	Anweisung
0000	C6 63		ADD A,63H

Ist das Ergebnis größer als 255, so wird es modulo 256 reduziert. Ob dieser Fall eintrat, erkennen wir nach Ausführung des Befehls am Zustand des Übertrag-Flags. Der ADD-Befehl setzt die Flags folgendermaßen (ein Flag ist gesetzt, wenn es den Inhalt 1 besitzt, rückgesetzt oder gelöscht, wenn es den Inhalt 0 hat):

S: gesetzt, falls Bit 7 des Ergebnisses gesetzt
Z: gesetzt, falls Ergebnis gleich Null
H: gesetzt, falls Übertrag von Bit 3
P: gesetzt, falls Überlauf auftrat
N: rückgesetzt
C: gesetzt, falls Übertrag von Bit 7

Wurde das Ergebnis also modulo 256 reduziert (das nicht reduzierte Ergebnis ist somit nicht als »Byte« darstellbar), so ist das Übertrag-Flag gesetzt, ansonsten ist es rückgesetzt.

Probieren wir also folgendes Beispiel, in dem eine solche Reduktion notwendig wird: C4H + 93H = 157H. Das zugehörige Programmstück lautet:

```
        ADD         A,93H       ; Inhalt des A-Registers
                                ; um den Wert 93H erhoehen
```

Vor seiner Ausführung müssen wir mit dem Debugger noch den Wert C4H ins A-Register bringen. Nach Ausführung des Programmstücks kontrollieren wir die Flags und das A-Register.

Merke: Der erste Operand des (8-Bit) ADD-Befehls ist stets das A-Register!

Ein Datenwert vom Typ »Byte« kann des weiteren auch interpretiert werden als Darstellung einer ganzen Zahl im Bereich −128 bis +127 (Grenzen eingeschlossen); die Darstellung erfolgt hierbei im 2-Komplement (siehe Kapitel 2). Ein Byte stellt dabei genau dann eine negative Zahl dar, wenn das Bit 7 gesetzt ist.

Wir addieren derart dargestellte Zahlen mit demselben ADD-Befehl. Kann das Ergebnis der Operation (vor einer Reduktion modulo 256) nicht als 2-Komplement einer ganzen Zahl im Bereich −128 bis +127 gedeutet werden, so liegt ein Überlauf vor. Bei Überlauf wird das Überlauf-Flag gesetzt.

Wir machen uns an folgenden Beispielen (mit Hilfe des Debuggers oder durch manuelle Berechnung) den Unterschied zwischen *Übertrag* und *Überlauf* klar:

FFH + 20H = ?
72H + 14H = ?
D4H + 9BH = ?
4FH + 22H = ?

Wir wollen nun den Inhalt des A-Registers (gedeutet als vorzeichenlose ganze Zahl) um eine Konstante vom Typ »Byte« vermindern, zum Beispiel um 35H. Formal

A <– <A> – 35H

Dies tun wir mit Hilfe des Befehls **SUB** (subtract):

```
        SUB     35H     ; Inhalt des A-Registers
                        ; um den Wert 35H vermindern
```

Der zugehörige Objekt-Code lautet:

Adresse	Objekt-Code	Marke	Anweisung
0000	D6 35		SUB 35H

Merke: Der erste Operand des SUB-Befehls ist stets das A-Register; es wird aber nicht explizit angegeben!

Wie schon beim ADD-Befehl wird auch beim SUB-Befehl das Ergebnis nötigenfalls modulo 256 reduziert. Ist der Wert im A-Register kleiner als der Wert des Arguments (beide als nicht-

negative ganze Zahlen interpretiert), so muß zur Subtraktion »geborgt« werden; es wird dann das Übertrag-Flag gesetzt.

Wieder können wir die Arithmetik auch auf ganze Zahlen in 2-Komplement-Darstellung anwenden. Ist das Ergebnis der Operation nicht als 2-Komplement einer ganzen Zahl im Bereich −128 bis +127 interpretierbar, so tritt ein Überlauf auf; es wird dann das Überlauf-Flag gesetzt.

Der SUB-Befehl setzt die Flags folgendermaßen:

S:	gesetzt, falls Bit 7 des Ergebnisses gesetzt
Z:	gesetzt, falls Ergebnis gleich Null
H:	rückgesetzt, falls Borgen von Bit 4 nötig war
P:	gesetzt, falls Überlauf auftrat
N:	gesetzt
C:	gesetzt, falls Borgen nötig war

Versuche nun folgende Beispiele im Debugger (achte dabei einmal auf alle Flags!):

37H - 37H = ?
62H - 7AH = ?
93H - 74H = ?
48H - A5H = ?

Für das Verständnis der Arithmetik ist es ganz wichtig, sich klar zu machen, daß ein und derselbe ADD- beziehungsweise SUB-Befehl, auf vorzeichenlose ganze Zahlen oder auf vorzeichenbehaftete ganze Zahlen in 2-Komplement-Darstellung angewandt, jeweils das richtige Ergebnis (in derselben Darstellung) liefert. Der Z80 kann dabei gar nicht wissen, was die Darstellung bedeuten soll, er führt immer ein und dasselbe Verfahren aus. Daß dies funktioniert, liegt an den Eigenschaften der 2-Komplement-Darstellung, die bezüglich Addition und Subtraktion mit der Darstellung vorzeichenloser ganzer Zahlen verträglich ist. Der Programmierer weiß natürlich, was die Darstellung bedeuten soll; an irgendeiner Stelle eines vollständigen Programms mit Ein-/Ausgabe wird von dieser Kenntnis mit Sicherheit auch Gebrauch gemacht. Es wird auf diese Weise also Information aus den Daten entfernt und im Programm versteckt (ein allgemeines Prinzip der Programmierung, insbesondere der maschinennahen Programmierung).

Mit Hilfe des Befehls **NEG** (negate) wird der Inhalt des A-Registers (gedeutet als 2-Komplement einer ganzen Zahl im Bereich −128 bis +127) negiert; das gleiche Ergebnis würde man erzielen, wenn man den Inhalt des A-Registers von Null subtrahieren würde:

A <– 0 – <A>

Der NEG-Befehl hat keine Argumente:

```
NEG                    ; Inhalt des A-Registers negieren
```

Als Objekt-Code ergibt sich:

Adresse	Objekt-Code	Marke	Anweisung
0000	ED 44		NEG

Die Flags werden folgendermaßen gesetzt:

S: gesetzt, falls Bit 7 des Ergebnisses gesetzt
Z: gesetzt, falls Ergebnis gleich Null
H: rückgesetzt, falls Borgen von Bit 4 nötig war
P: gesetzt, falls Überlauf auftrat
N: gesetzt
C: rückgesetzt, falls Ergebnis gleich Null

Ist der Inhalt des A-Registers vor der Operation 80H (das ist die 2-Komplement-Darstellung von −128), so tritt ein Überlauf auf, weil das Ergebnis der Operation +128 ist und somit nicht in einem Byte als 2-Komplement dargestellt werden kann; es wird modulo 256 reduziert, als Ergebnis erhält man 80H.

Merke: Der NEG-Befehl bezieht sich immer auf das A-Register!

Ist das Resultat eines ADD-Befehls, SUB-Befehls oder NEG-Befehls negativ (nach einer eventuellen Reduktion modulo 256), das heißt, hat Bit 7 den Wert 1, so wird das Vorzeichen-Flag gesetzt.

Eine Bemerkung zu den Beispielen und Übungen: Die Werte sind so gewählt, daß möglichst unterschiedliche Kombinationen der Flags sichtbar werden.

Übungen

1. Führe folgende Additionen durch Programme aus:
   41H + 20H
   3 + 48
   254Q + 221Q
   1110110B + 11010010B

2. Führe folgende Subtraktionen durch Programme aus:

   68H - 20H
   57 - 48
   231Q - 116Q
   00110001B - 10100001B

3. Negiere folgende Zahlen durch Programme:

   80H
   0
   164Q
   10111001B

4. Versuche, für jeden der gezeigten Befehle und für jede Kombination von Zuständen der Flags geeignete Operanden zu finden! Für welche Kombinationen geht es nicht? Warum nicht?

7
Zeichen

Unter einem Zeichen (engl. character) versteht man Buchstaben, Ziffern, das Leerzeichen (engl. space oder blank, wird meist mit SP abgekürzt, nicht zu verwechseln mit der Bezeichnung des Stapelzeigers), Sonderzeichen wie Punkt, Komma und Klammern, sowie Steuerzeichen. Steuerzeichen dienen zur Positionierung von Ausgabegeräten und zur Kommunikation zwischen Prozessor und Ein-/Ausgabe-Geräten.

7.1 Der ASCII-Code

Eine gebräuchliche Codierung für Zeichen ist der ASCII-Code (American Standard Code for Information Interchange). Der ASCII-Code ist ein 7-Bit-Code, das heißt zur Codierung eines Zeichens werden 7 Bits verwendet. Da der Z80 ein Byte-orientierter Prozessor ist, wird zur Darstellung eines Zeichens in ASCII normalerweise ein Byte verwendet. Bit 7 hat gewöhnlich entweder stets den Wert 0 oder (seltener) stets den Wert 1; zu Zwecken der Datensicherung wird manchmal auch dieses Bit so gesetzt, daß die Gesamtzahl der Bits mit Wert 1 in einem Byte – die Parität – gerade (engl. parity even) oder ungerade (engl. parity odd) ist.

Die Codierung der einzelnen Zeichen entnehme man folgender Tabelle (wir nehmen dabei an, daß Bit 7 stets rückgesetzt ist):

Tabelle 7.1. *Der ASCII-Code*

00H	NUL	20H	SP	04H	EOT	24H	$
01H	SOH	21H	!	05H	ENQ	25H	%
02H	STX	22H	"	06H	ACK	26H	&
03H	ETX	23H	#	07H	BEL	27H	'

08H	BS	28H	(	44H	D	64H	d
09H	HT	29H	)	45H	E	65H	e
0AH	LF	2AH	*	46H	F	66H	f
0BH	VT	2BH	+	47H	G	67H	g
0CH	FF	2CH	,	48H	H	68H	h
0DH	CR	2DH	–	49H	I	69H	i
0EH	SO	2EH	.	4AH	J	6AH	j
0FH	SI	2FH	/	4BH	K	6BH	k
10H	DLE	30H	0	4CH	L	6CH	l
11H	DC1	31H	1	4DH	M	6DH	m
12H	DC2	32H	2	4EH	N	6EH	n
13H	DC3	33H	3	4FH	O	6FH	o
14H	DC4	34H	4	50H	P	70H	p
15H	NAK	35H	5	51H	Q	71H	q
16H	SYN	36H	6	52H	R	72H	r
17H	ETB	37H	7	53H	S	73H	s
18H	CAN	38H	8	54H	T	74H	t
19H	EM	39H	9	55H	U	75H	u
1AH	SUB	3AH	:	56H	V	76H	v
1BH	ESC	3BH	;	57H	W	77H	w
1CH	FS	3CH	<	58H	X	78H	x
1DH	GS	3DH	=	59H	Y	79H	y
1EH	RS	3EH	>	5AH	Z	7AH	z
1FH	VS	3FH	?	5BH	[	7BH	{
40H	@	60H	`	5CH		7CH	\
41H	A	61H	a	5DH	]	7DH	}
42H	B	62H	b	5EH	^	7EH	~
43H	C	63H	c	5FH	_	7FH	DEL

Die Zeichen mit den Codierungen 00H bis 1FH sowie 7FH sind Steuerzeichen. Der Prozessor unterscheidet nicht zwischen Steuerzeichen und sichtbaren Zeichen; dies tun nur die Ein-/Ausgabe-Geräte.

Die Bedeutung mancher Steuerzeichen variiert je nach Ein-/Ausgabe-Gerät. Feste Bedeutung auf fast allen Geräten haben folgende Zeichen:

Tabelle 7.2. *Einige Steuerzeichen des ASCII-Codes*

Zeichen	Bedeutung	Funktion
BEL	bell	Klingel (Summer, Piepser) ertönen lassen
BS	backspace	auf vorhergehendes Zeichen zurückpositionieren
HT	horizontal tab	auf nächsten Tabulator der Zeile positionieren

Zeichen	Bedeutung	Funktion
LF	line feed	eine Zeile tiefer gehen
FF	form feed	auf den Anfang der nächsten Seite positionieren
CR	carriage return	auf den Anfang derselben Zeile positionieren

Für die Steuerung flexibler Ausgabegeräte reichen die wenigen Steuerzeichen nicht aus. Man behilft sich damit, daß Steuerfunktionen solcher Geräte durch eine Folge von Zeichen ausgelöst werden, die mit dem Zeichen ESC (escape) beginnt, sonst aber beliebige (auch sichtbare) Zeichen enthalten darf.

Um in verschiedenen wichtigen Sprachen einen der Sprache angepaßten Zeichensatz zur Verfügung zu haben, gibt es Varianten des ASCII-Codes. Nachfolgend einige Beispiele:

Tabelle 7.3. *Nationale und internationale Varianten des ASCII-Codes*

	23H	24H	40H	5BH	5CH	5DH	5EH	60H	7BH	7CH	7DH	7EH
ASCII	#	$	@	[	\	]	^	`	{	|	}	~
Deutsch	#	$	§	Ä	Ö	Ü	^	`	ä	ö	ü	ß
Schwedisch, Finnisch	§	¤	É	Ä	Ö	Å	^	é	ä	ö	å	ü
Dänisch, Norwegisch	#	¤	É	Æ	Ø	Å	Ü	é	æ	ø	å	ü
Britisch	£	$	@	[	\	]	^	`	{	|	}	¯
Französisch, Belgisch	£	$	à	°	ç	§	^	`	é	ù	è	¨
International	#	¤	@	[	\	]	^	`	{	|	}	¯

Außer dem ASCII-Code werden gelegentlich auch andere Codes wie Baudot-Code (Fernschreib-Code) oder EBCDIC (Extended Binary Coded Decimal Interchange Code) verwendet. Erweiterungen des ASCII-Codes zum 8-Bit Code durch Hinzunahme von Graphikzeichen sind gebräuchlich (IBM-Code). Gelegentlich kommt es vor, daß Ein-/Ausgabe-Geräte keine Kleinbuchstaben oder keine Großbuchstaben verarbeiten können. Diese Buchstaben sind dann im Code sinngemäß durch andere ersetzt.

Übungen

1. Was bedeutet folgende ASCII-Codierung einer Zeichenfolge?

47H 75H 74H 20H 67H 65H 6DH 61H 63H 68H 74H 21H 0DH 0AH

2. Codiere folgenden Satz in ASCII:

14% Mehrwertsteuer von 120.– DM, das sind 16.80 DM.

7.2 Manipulation von Zeichen

ASCII-codierte Zeichen unterscheiden sich im Speicher nicht von numerischen Daten des Types »Byte« und werden deshalb vom Prozessor auch wie solche behandelt.

Der Assembler bietet uns die Möglichkeit, Konstanten vom Typ »Zeichen« in der ASCII-Darstellung oder in einer der numerischen Zahl-Darstellungen zu notieren. Die Befehle

```
        LD          A,3FH       ; Lade A-Register
                                ; mit dem Wert 3FH
```

und

```
        LD          A,'?'       ; Lade A-Register mit dem
                                ; ASCII-Code des Fragezeichens
```

werden vom Assembler beide auf den gleichen Objekt-Code abgebildet, nämlich

Adresse	Objekt-Code	Marke	Anweisung	
0000	3E 3F		LD	A,'?'

Merke: Im Programmtext müssen Zeichenkonstanten in einfache Hochkommas eingeschlossen werden.

Da der Z80 mit ASCII-codierten Zeichen verfährt wie mit numerischen Daten vom Typ »Byte«, kann man auf ihnen arithmetische Operationen ausführen. Besonders praktisch ist dies bei den Dezimalziffern, deren Codierung eine lückenlos aufsteigende Folge bildet.

Beispiel: Das A-Register enthalte den numerischen Wert einer Dezimalziffer (zum Beispiel 03H). Wir wollen diesen in die entsprechende ASCII-Codierung der Dezimalziffer umwandeln.

Die ASCII-Codierung einer Dezimalziffer erhalten wir, indem wir 30H zum numerischen Wert der Ziffer addieren:

```
        ADD         A,30H       ; binaer codierte Dezimalziffer
                                ; in ASCII umwandeln
```

Führe nun die Umrechnung mit Hilfe des Programms durch! Siehe dazu auch Aufgabe 1 aus Kapitel 6.2!

Analog die Umkehrung: Das A-Register enthalte die ASCII-Codierung einer Dezimalziffer (zum Beispiel 39H für die Ziffer »9«). Wir wollen diese in ihren numerischen Wert umwandeln.

Den numerischen Wert einer ASCII-codierten Dezimalziffer erhalten wir, indem wir 30H von der Ziffer subtrahieren:

```
SUB     30H     ; ASCII-codierte Dezimalziffer
                ; in numerischen Wert umwandeln
```

Führe die Umwandlung mittels Programm aus! Vergleiche mit Aufgabe 2 aus Kapitel 6.2!

Der Assembler erzeugt denselben Objekt-Code, wenn wir statt dessen – etwas besser dokumentierend – schreiben:

```
SUB     '0'     ; ASCII-codierte Dezimalziffer
                ; in numerischen Wert umwandeln.
                ; '0' geht ueber in 0.
                ; Dezimalziffern sind sowohl
                ; in Hex wie in ASCII
                ; fortlaufend aufsteigend.
```

Eine Zeichenkonstante als Argument eines Befehls wird vom Assembler durch ihre ASCII-Codierung (vom Typ »Byte«) ersetzt.

Genauso einfach wie die Umwandlung von Ziffern in Zahlen ist die Umwandlung von Großbuchstaben in Kleinbuchstaben. Sowohl Großbuchstaben wie Kleinbuchstaben sind jeweils als lückenlose aufsteigende Folge codiert.

Beispiel: Im A-Register stehe in ASCII-Codierung ein Großbuchstabe (zum Beispiel 41H für den Buchstaben »A«). Wir wandeln diesen in den entsprechenden Kleinbuchstaben um.

Aus einem ASCII-codierten Großbuchstaben entsteht der entsprechende Kleinbuchstabe durch Addition von 20H:

```
ADD     A,20H   ; Grossbuchstaben in
                ; Kleinbuchstaben umwandeln
```

Führe die Umrechnung durch und vergleiche mit Aufgabe 1 aus Kapitel 6.2!

Oder umgekehrt: Im A-Register stehe ein ASCII-codierter Kleinbuchstabe (zum Beispiel 68H für den Buchstaben »h«). Wandle diesen in den entsprechenden Großbuchstaben um!

Aus einem ASCII-codierten Kleinbuchstaben entsteht der entsprechende Großbuchstabe durch Subtraktion von 20H:

```
SUB     20H     ; Kleinbuchstaben in
                ; Grossbuchstaben umwandeln
```

Führe die Umwandlung per Programm durch; vergleiche auch mit Aufgabe 2 aus Kapitel 6.2!

Wir können uns auch vorstellen, daß wir die Großbuchstaben beziehungsweise Kleinbuchstaben von 0 bis 25 (bei deutschem Zeichensatz bis 28) fortlaufend numeriert haben und nun zuerst aus dem Kleinbuchstaben die zugeordnete Nummer bestimmen, aus dieser dann den entsprechenden Großbuchstaben. Formal

A <– <A> – 'a'
A <– <A> + 'A'

Dazu benötigen wir wie bisher nur einen einzigen Befehl:

```
        SUB     'a' - 'A'     ; Kleinbuchstaben in
                              ; Grossbuchstaben umwandeln
```

Der Objekt-Code dieses Befehls unterscheidet sich nicht von dem des vorhergehenden Befehls:

Adresse	Objekt-Code	Marke	Anweisung	
0000	D6 20		SUB	'a' – 'A'

Findet der Assembler für ein Argument eines Befehls statt einer Konstanten einen Ausdruck, so berechnet er den Wert des Ausdrucks und verwendet diesen als Argument. Zeichenkonstanten werden dabei durch ihre ASCII-Codierungen ersetzt. Bei numerischen Operationen wird unterstellt, daß die Operanden im 2-Komplement dargestellt sind. Die Berechnung wird modulo $2^{\text{Argumentlänge}}$ durchgeführt, damit das Ergebnis eine der Länge des Arguments (meist 8 Bits oder 16 Bits) angepaßte Größe besitzt.

Teste alle angegebenen Umrechnungen mit weiteren Daten im Debugger aus!

Übungen

1. Schreibe ein Programm, das einen Großbuchstaben auf seine Ordnungszahl abbildet. Dabei soll gelten: ord('A') = 1, ord('B') = 2, ...

2. Schreibe ein Programm, das einen Kleinbuchstaben aus seiner Ordnungszahl herstellt. Dabei soll gelten: ord('a') = 1, ord('b') = 2, ...

8
Sequenzen

Eine Sequenz ist eine Folge von Befehlen, die in einer vorgeschriebenen Reihenfolge vom Prozessor verarbeitet werden.

Um dem Assembler die gewünschte Reihenfolge der Befehle mitzuteilen, schreiben wir diese - jeden Befehl in eine eigene Zeile - von oben nach unten fortlaufend auf.

Wollen wir zum Beispiel den Wert 48H ins A-Register laden und dann den Wert 20H dazu addieren, so schreiben wir folgende Sequenz:

```
LD      A,48H     ; lade 48H ins A-Register
ADD     A,20H     ; erhöhe Inhalt des
                  ; A-Registers um 20H
```

Der zugehörige Objekt-Code sieht folgendermaßen aus:

Adresse	Objekt-Code	Marke	Anweisung	
0000	3E 48		LD	A,48H
0002	C6 20		ADD	A,20H

8.1 Verwendung von Variablen

Bis jetzt haben wir unsere Datenwerte stets direkt als Argumente von Assemblerbefehlen angegeben (Adressierungsart: immediate). Dies entspricht der Verwendung von Konstanten in höheren Programmiersprachen. Nun wollen wir die Realisierung von Variablen kennenlernen.

Eine Variable vom Typ »Byte« wird durch Angabe einer Adresse spezifiziert. Der Inhalt der adressierten Speicherzelle ist der Variableninhalt (wir werden später auch Datentypen kennenlernen, die mehr Speicher benötigen als ein Byte; die Variablenadresse gibt dann die Anfangsadresse eines zusammenhängenden Speicherbereichs an, in welchem der Inhalt der Variablen abgelegt wird).

Wir nehmen nun an, daß unsere Variable unter der Adresse 540EH zu finden ist. Als erstes schreiben wir (zum Beispiel mit Hilfe des Debuggers oder mit einem POKE-Befehl eines BASIC-Programms) einen Datenwert vom Typ »Byte« in die Speicherzelle. Jetzt können wir den Inhalt der Variablen in das A-Register laden. Dies geschieht mit Hilfe des bereits bekannten LD-Befehls. Allerdings geben wir den Datenwert jetzt nicht direkt, sondern in Form seiner Adresse an (Adressierungsart: direct). Diese wird in runde Klammern eingeschlossen, um anzuzeigen, daß das Argument des Befehls die Adresse des Datenwerts ist (was für den Prozessor soviel heißt wie: ersetze das Argument durch den Inhalt der adressierten Speicherzelle):

```
LD          A,(540EH)  ; lade Inhalt der Speicherzelle
                       ; mit der Adresse 540EH
                       ; ins A-Register
```

Der zugehörige Objekt-Code lautet:

Adresse	Objekt-Code	Marke	Anweisung	
0000	3A 0E 54		LD	A,(540EH)

Studiere diesen Objekt-Code etwas länger, um zu verstehen, wie der Assembler Adressen im Befehl plaziert!

Umgekehrt können wir auch den Inhalt des A-Registers in eine Variable (zum Beispiel mit der Adresse 540FH) schreiben:

```
LD          (540FH),A  ; lade Inhalt des A-Registers
                       ; in die Speicherzelle
                       ; mit der Adresse 540FH
```

Um die Wirkung unseres Befehls zu überprüfen, müssen wir den Speicherinhalt der Adresse 540FH inspizieren. Dies tun wir zum Beispiel mit dem Debugger oder mittels der BASIC-Funktion PEEK.

Merke: Im Assemblerprogramm bedeutet eine von runden Klammern eingeschlossene Adresse den Inhalt der Speicherzelle mit dieser Adresse!

Spielen Sie jetzt ein bißchen mit den Assemblerbefehlen, die Sie bisher kennengelernt haben!

Wir schreiben nun unser erstes längeres Programm: Addiere 24H zu dem unter der Adresse 534BH abgelegten Datenwert vom Typ »Byte« (unter einer Adresse ablegen bedeutet: in der Speicherzelle mit dieser Adresse ablegen)!

Dazu laden wir den Datenwert aus der Speicherzelle mit der Adresse 534BH ins A-Register, addieren 24H, und schreiben den Wert anschließend wieder in die Speicherzelle mit der Adresse 534BH zurück. Der Algorithmus dazu lautet formal:

A <– <(534BH)>
A <– <A> + 24H
(534BH) <– <A>

Wir geben zur Abwechslung auch einmal das Flußdiagramm dazu an (Flußdiagramme ohne Fallunterscheidungen und Unterprogramme sind jedoch recht fade Gesellen):

Bild 8.1. *Erhöhen des Inhalts einer Speicherzelle*

Unser Programm lautet nun:

```
LD      A,(534BH)   ; erhöhe Inhalt der Variablen
ADD     A,24H       ; mit der Adresse 534BH
LD      (534BH),A   ; um den Wert 24H
```

Nun ergibt sich schon ein etwas komplizierteres Objekt-Programm:

Adresse	Objekt-Code	Marke	Anweisung	
0000	3A 4B 53		LD	A,(534BH)
0003	C6 24		ADD	A,24H
0005	32 4B 53		LD	(534BH),A

Übungen

1. Der Inhalt der Speicherzelle mit der Adresse 7422H soll um 17 erniedrigt werden.

2. Negiere den Inhalt der Speicherzelle mit der Adresse 5444H.

3. Für Tüftler: Schreibe ein Programm, das einen im A-Register stehenden Großbuchstaben in den symmetrisch gelegenen Großbuchstaben umwandelt, also

   A in Z
   B in Y
   :
   :
   :
   Z in A

8.2 Einfache Multiplikationsprogramme

Wir kommen nun zu einer wichtigen Anwendung unseres bisherigen Könnens: Multiplikation des Inhalts des A-Registers (interpretiert als vorzeichenlose ganze Zahl) mit einer positiven ganzzahligen Konstanten; wir wollen dabei stets annehmen, daß das Ergebnis klein genug ist, um wieder im A-Register untergebracht zu werden.

Als Vorbereitung lernen wir die Multiplikation des A-Registers mit der Konstanten 2 kennen. Dies ist einfach die Addition des A-Registers zu sich selbst:

A <– <A> + <A>

Wir verwenden dazu den bereits bekannten ADD-Befehl:

```
ADD     A,A     ; verdopple Inhalt
                ; des A-Registers
```

Als zweites Argument des ADD-Befehls kann statt einer Konstanten auch eines der Register A, B, C, D, E, H, L stehen; der Inhalt des A-Registers wird dann um den Inhalt dieses zweiten Registers (das ebenfalls das A-Register sein kann) erhöht.

Versuche auch folgendes Beispiel

```
ADD     A,B     ; erhoehe Inhalt des A-Registers
                ; um Inhalt des B-Registers
```

Als nächstes folgt die Multiplikation des A-Registers mit der Konstanten 3. Wegen der Darstellung $3 = 1 + 2 * 1$ verdoppeln wir zuerst den Inhalt des A-Registers und addieren dann den ursprünglichen Inhalt nochmals dazu; diesen müssen wir dazu aber in einem anderen 8-Bit-Register hilfsweise ablegen, da er nach der Verdopplung des A-Registers sonst nicht mehr zur Verfügung steht.

Mit dem LD-Befehl kann man auch den Inhalt eines der Register A, B, C, D, E, H, L in ein anderes abspeichern. Wir verwenden beispielsweise das D-Register:

D <– <A>
A <– 2 * <A>
A <– <A> + <D>

Das zugehörige Programm lautet:

```
LD      D,A     ; Operand sichern
ADD     A,A     ; Operand verdoppeln
ADD     A,D     ; Operand verdreifachen
```

Teste das Beispiel mit selbstgewählten Daten aus!

Kommen wir nun zum allgemeinen Fall: Multiplikation des Inhalts des A-Registers mit einer beliebigen positiven ganzzahligen Konstanten n.

Wir schreiben dazu die Binärdarstellung von n auf, $n = n_s n_{s-1} ... n_0$. Damit schreiben wir n (im sogenannten »Horner-Schema«) als $n = n_0 + 2 * (n_1 + 2 * (n_2 + ... + 2 * (n_{s-1} + 2 * n_s) ...))$.

Der Algorithmus zur Multiplikation lautet nun:

Hilfsregister <– <A>
wiederhole
    A <– 2 * <A>
    wenn $n_i=1$
    dann A <– <A> + <Hilfsregister>
mit i von s–1 **bis** 0 **in Schritten von** –1

Da n fest vorgegeben ist, kommt im jeweiligen Programm weder eine Schleife noch eine Verzweigung vor; wir lösen den Algorithmus per Hand in die entsprechende Folge arithmetischer Befehle auf. Diese Form der Multiplikation nennt man deshalb auch *gestreckte Multiplikation.*

Beispiel: n = 13. Es ist 13 = 1101B = 1 + 2 * (0 + 2 * (1 + 2 * 1)). Also erhalten wir folgendes Programm (mit dem D-Register als Hilfsregister):

```
LD      D,A     ; Operand sichern
ADD     A,A     ; Operand verdoppeln
ADD     A,D     ; Operand verdreifachen
ADD     A,A     ; Operand versechsfachen
ADD     A,A     ; Operand verzwoelffachen
ADD     A,D     ; Operand verdreizehnfachen
```

Wenn - wie angenommen - das Endergebnis im A-Register Platz findet, so gilt dies auch für jedes Zwischenergebnis. Andernfalls fällt bei mindestens einem der vorkommenden Additions-Befehle ein Übertrag an (nicht unbedingt bei der letzten Addition: Überlege, was bei <A>=70 und n=5 passiert!).

Der beschriebene Algorithmus wird häufig für n=10 verwendet zur Umrechnung von Folgen von Dezimalziffern in binär codierte Zahlen.

An einem weiteren Beispiel wollen wir den Begriff »Effizienz« etwas näher erläutern. Wir wenden den Multiplikations-Algorithmus für n = 15 an:
15 = 1111B = 1 + 2 * (1 + 2 * (1 + 2 * 1)). Somit erhalten wir folgendes Programm

```
LD      D,A     ; Operand sichern
ADD     A,A     ; Operand verdoppeln
ADD     A,D     ; Operand verdreifachen
ADD     A,A     ; Operand versechsfachen
ADD     A,D     ; Operand versiebenfachen
ADD     A,A     ; Operand vervierzehnfachen
ADD     A,D     ; Operand verfuenfzehnfachen
```

mit dem Objekt-Code

```
Adresse  Objekt-Code  Marke  Anweisung
0000     57                  LD      D,A
0001     87                  ADD     A,A
0002     82                  ADD     A,D
0003     87                  ADD     A,A
0004     82                  ADD     A,D
0005     87                  ADD     A,A
0006     82                  ADD     A,D
```

Wir schreiben nun ein weiteres Programm zur Lösung der gleichen Aufgabe; dazu verwenden wir allerdings auch den SUB-Befehl, der wie der ADD-Befehl ebenfalls ein 8-Bit-Register als Argument haben kann. Wegen $15 = 2 * 2 * 2 * 2 * 1 - 1$ können wir schreiben:

```
LD      D,A     ; Operand sichern
ADD     A,A     ; Operand verdoppeln
ADD     A,A     ; Operand vervierfachen
ADD     A,A     ; Operand verachtfachen
ADD     A,A     ; Operand versechzehnfachen
SUB     D       ; Operand verfuenfzehnfachen
```

Dieses Programm besitzt den Objekt-Code

Adresse	Objekt-Code	Marke	Anweisung	
0000	57		LD	D,A
0001	87		ADD	A,A
0002	87		ADD	A,A
0003	87		ADD	A,A
0004	87		ADD	A,A
0005	92		SUB	D

Das zweite Programm hat zwei ADD-Befehle weniger als das vorhergehende, dafür ist ein SUB-Befehl hinzugekommen. Da ein SUB-Befehl genauso lange zur Ausführung braucht wie ein ADD-Befehl (siehe Anhang B) und der erzeugte Objekt-Code der beiden Befehle jeweils ein Byte lang ist, läuft das zweite Programm schneller (Rechenzeit-Effizienz) und ist obendrein auch noch kürzer (Speicher-Effizienz).

Die beiden Programme unterscheiden sich auch noch in einer anderen Hinsicht: Beim zweiten Programm kann es vorkommen, daß das Endergebnis im A-Register Platz hat, nicht aber das letzte Zwischenergebnis; es fällt dann ein Übertrag an. In jedem Fall steht aber bei beiden Algorithmen als Endergebnis das Produkt (modulo 256) im A-Register.

Teste alle vorgekommenen Beispiele mit brauchbaren Daten aus; wenn Dein Debugger das Einzelschritt-Verfahren zuläßt, so sieh Dir auch die einzelnen Schritte in ihrer Wirkung genau an!

Übungen

1. Schreibe ein Programm, welches das Sechzehnfache des Inhalts des C-Registers ins A-Register schreibt.

2. Der Inhalt des A-Registers soll verdreißigfacht werden. Schreibe eine geeignete Routine und achte dabei auf Effizienz.

3. Für welche Daten ergibt sich bei der Verfünfzehnfachung mittels SUB-Befehls ein Überlauf während der Berechnung, für das Verfahren mit den ADD-Befehlen jedoch nicht?

4. Schreibe ein Programm, das unter Verwendung des A-Registers die Inhalte des B-Registers und D-Registers vertauscht.

9
Verzweigungen

Eine Verzweigung ist eine Stelle im Programm, an welcher der weitere Programmablauf in Abhängigkeit von einer Bedingung gesteuert wird.

9.1 Einseitige Verzweigungen

Bei einer einseitigen Verzweigung wird ein bestimmtes Programmstück nur dann ausgeführt, wenn eine gewisse Bedingung erfüllt ist. Ansonsten wird das Programmstück übersprungen. Formal ausgedrückt:

wenn Bedingung erfüllt
dann Programmstück ausführen

Betrachten wir folgendes Problem: Im A-Register und im C-Register stehe je eine ganze Zahl in 2-Komplement-Darstellung. Die Summe dieser beiden Zahlen soll ins A-Register geschrieben werden. Bei der Addition kann ein Überlauf auftreten; in diesem Fall soll das A-Register den Wert 0 erhalten.

Wir addieren also den Inhalt des C-Registers mit dem ADD-Befehl zum Inhalt des A-Registers. Ob dabei ein Überlauf auftritt, können wir anschließend am Zustand des Überlauf-Flags erkennen. Wenn das Überlauf-Flag gesetzt ist (das heißt den Inhalt 1 hat), laden wir den Wert 0 ins A-Register:

wenn Überlauf-Flag gesetzt
dann lade 0 ins A-Register

Ist das Überlauf-Flag nicht gesetzt, so überspringen wir den Lade-Befehl:

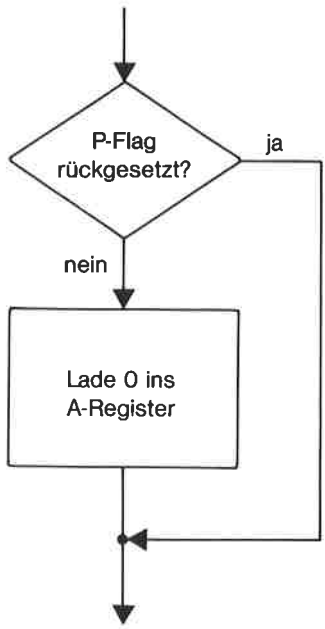


Bild 9.1. *Flußdiagramm: Beispiel einer einseitigen Verzweigung*

Wir führen also einen *bedingten Sprung* aus, und zwar mit Hilfe des Befehls **JP** (jump):

```
        ADD     A,C         ; Summe bilden
        JP      PO,KUEBER   ; springe, wenn kein Ueberlauf
        LD      A,0         ; bei Ueberlauf lade 0
KUEBER: NOP                 ; gemeinsame Fortsetzungsstelle
```

Als Objekt-Code wird daraus erzeugt:

Adresse	Objekt-Code	Marke	Anweisung	
0000	81		ADD	A,C
0001	E2 06 00		JP	PO,KUEBER
0004	3E 00		LD	A,0
0006	00	KUEBER:	NOP	

Das erste Argument des JP-Befehls gibt die Bedingung für den Sprung an. In unserem Fall ist das PO (parity odd); eine Begründung für diese spezielle Bezeichnung werden wir später noch kennenlernen. PO bedeutet, daß das Überlauf-Flag nicht gesetzt ist, bei unserer Addition also kein Überlauf auftrat.

Beachte, daß die Bedingung des Sprungs die Negation der Bedingung unseres ursprünglichen Algorithmus ist! Dies hängt damit zusammen, daß im Algorithmus – wie er in der Beschreibungssprache formuliert ist – die Bedingung die Ausführung einer Aktion regelt, während dagegen die Bedingung im Flußdiagramm – die unserer Realisierung durch das Programm mehr entspricht – das Überspringen der Aktion steuert.

Das zweite Argument des JP-Befehls ist die Adresse des Befehls, mit dem fortgefahren werden soll, wenn die Bedingung »PO« erfüllt ist. Statt diese Adresse auszurechnen und als Zahl anzugeben, schreiben wir vor den entsprechenden Befehl eine symbolische Adresse (eine Marke) und geben diese als Ziel des Sprungbefehls an. Der Assembler berechnet sich dann selbst die Speicheradresse, an welcher der Objekt-Code des anzuspringenden Befehls beginnt.

Es empfiehlt sich, als symbolische Adresse eine Bezeichnung zu wählen, die mnemotechnisch etwas über das zugrunde liegende Problem aussagt (»KUEBER« steht für »kein Überlauf«).

Nach der symbolischen Adresse KUEBER folgt in unserem Beispiel ein neuer Assemblerbefehl: **NOP** (no operation). Der NOP-Befehl führt tatsächlich keine Aktionen durch; er wird meist verwendet, um in Programmen Platz für spätere Erweiterungen oder Veränderungen vorzusehen (der Objekt-Code des NOP-Befehls belegt ein Byte). In unserem Beispiel wird der NOP-Befehl nur verwendet, um ein definiertes Sprungziel zu schaffen; er ist für den Sprung nicht essentiell. Ist der Sprung samt Sprungziel in ein größeres Programm eingebettet, so wird an Stelle des NOP-Befehls derjenige Befehl stehen, mit dem wir nach dem Sprung fortfahren wollen.

Wir werden nun unsere ersten beiden Pseudo-Operationen kennenlernen: **ORG** (origin) und **END** (end).

Mit der Pseudo-Operation ORG kann der Programmierer die Anfangsadresse des Programms festlegen; der ORG-Befehl hat als Argument eben diese Adresse, zum Beispiel

```
        ORG     4200H       ; Objekt-Code soll bei
                            ; Adresse 4200H beginnen
```

Enthält ein Programm keine ORG-Pseudo-Operation, so setzt der Assembler meist eine willkürliche Anfangsadresse fest (häufig 0000H). Es gibt allerdings auch Assembler, die Programme ohne ORG-Pseudo-Operation nicht akzeptieren.

Am Ende des Programms muß eine END-Pseudo-Operation stehen. Diese zeigt dem Assembler an, daß der Quelltext zu Ende ist. Jedes Programm enthält daher genau eine END-Pseudo-Operation. Fehlt diese, so erfolgt eine Warnung; als Ende wird das physikalische Ende des Quelltexts verwendet. Unser Programm sieht also folgendermaßen aus:

```
        ORG     4200H
        ADD     A,C         ; Summe bilden
        JP      PO,KUEBER   ; springe, wenn kein Ueberlauf
        LD      A,0         ; bei Ueberlauf lade 0
KUEBER: NOP                 ; gemeinsame Fortsetzungsstelle

        END
```

Für die Pseudo-Operationen selbst wird kein Objekt-Code erzeugt, obwohl sie die Form desselben mitbestimmen:

Adresse	Objekt-Code	Marke	Anweisung	
			ORG	4200H
4200	81		ADD	A,C
4201	E2 06 42		JP	PO,KUEBER
4204	3E 00		LD	A,0
4206	00	KUEBER:	NOP	
			END	

Wir werden unsere Beispielprogramme im allgemeinen ohne ORG-und END-Pseudo-Operationen aufschreiben. Zum Testen müssen Sie diese nach den Vorschriften Ihres Assemblers ergänzen!

Teste nun das Programm mit folgenden Werten aus (achte dabei auf die Flags!):

<A>=FFH <C>=3EH (LD-Befehl wird übersprungen)
<A>=84H <C>=9BH (LD-Befehl wird nicht übersprungen)

Wollen wir die Korrektheit einer einseitigen Verzweigung (also eines bedingten Sprungbefehls) testen, so müssen die Daten stets so gewählt werden, daß im einen Fall die entsprechende Bedingung des Algorithmus erfüllt ist, im andern Fall jedoch nicht. Dies ist aber nur eine Minimalanforderung; im Prinzip müßten wir alle möglichen Fälle ausprobieren (oder formal die Korrektheit beweisen), was natürlich zu aufwendig, bei umfangreichen Problemen sogar meist unmöglich ist.

Denke Dir weitere Beispiele für die Eingabedaten aus und teste damit das Programm!

Merke: Eine einseitige Verzweigung wird durch einen bedingten Sprung realisiert!

Wir werden nun weitere Formen von bedingten Sprüngen kennenlernen, die sich jeweils durch die verwendete Bedingung voneinander unterscheiden.

Zu lösen sei folgendes Problem: Im A-Register und im E-Register stehe je eine ganze Zahl in 2-Komplement-Darstellung. Im C-Register stehe −1 (ebenfalls in 2-Komplement-Darstellung). Subtrahiere den Inhalt des E-Registers vom Inhalt des A-Registers und setze das C-Register zu 0, falls kein Überlauf dabei auftritt (wir können mit dieser Technik an einer beliebigen anderen Stelle des Programms am Inhalt des C-Registers erkennen, ob bei der Subtraktion ein Überlauf auftrat oder nicht). Schreibe das Ergebnis der Subtraktion wieder ins A-Register.

In formaler Notation lautet unser Algorithmus:

A <– <A> – <E>
wenn <P> = 0
dann C <– 0

Wissen Sie noch, daß »P« dabei für das Überlauf-Flag steht?

Als Flußdiagramm dargestellt sieht der Algorithmus folgendermaßen aus:

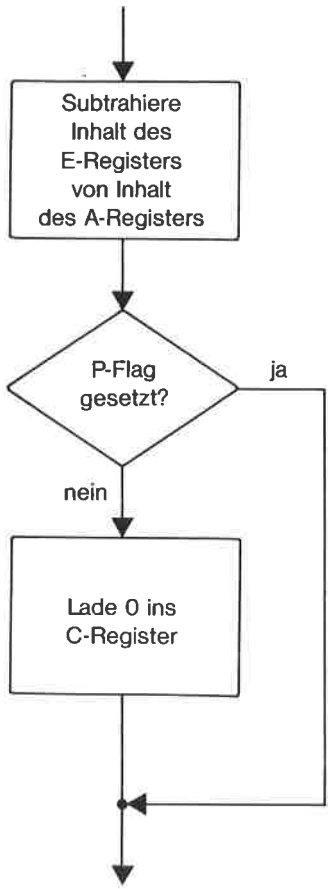


Bild 9.2. *Flußdiagramm: Notieren eines Ereignisses*

Als Bedingung verwenden wir diesmal PE (parity even), das ist die Negation von PO:

```
        SUB     E           ; Differenz bilden
        JP      PE,UEBERL   ; springe, wenn Ueberlauf
        LD      C,0         ; notiere Ereignis
UEBERL: NOP                 ; gemeinsame Fortsetzungsstelle
```

Nächstes Problem: Im A-Register und im B-Register stehe je eine ganze Zahl in 2-Komplement-Darstellung. Subtrahiere den Inhalt des B-Registers vom Inhalt des A-Registers. Sollte das Ergebnis negativ sein, so bilde den absoluten Betrag davon. Das Endergebnis soll wieder im A-Register stehen. Eventuell auftretende Überläufe wollen wir vorläufig vernachlässigen.

Zunächst wieder die Formalisierung des Problems (denke daran, daß »S« für das Vorzeichen-Flag steht!):

A <– <A> – <B>
wenn <S> = 1
dann A <– 0 – <A>

Und das zugehörige Flußdiagramm:

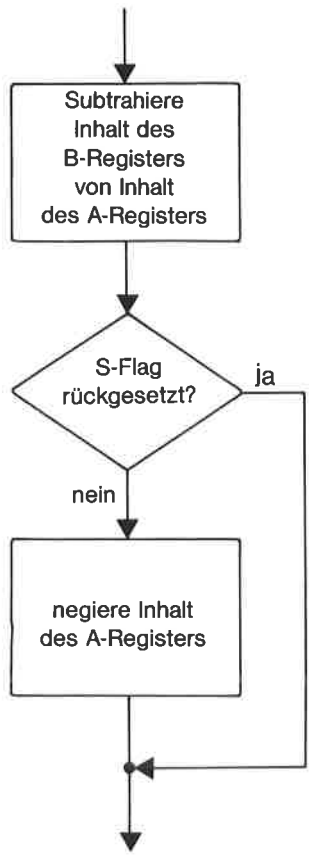


Bild 9.3. *Flußdiagramm: Absoluter Betrag*

Die geeignete Bedingung für dieses Beispiel ist P (plus), das heißt bei positivem Vorzeichen wird ein Sprung ausgeführt (die Null besitzt übrigens auch ein positives Vorzeichen!):

```
SUB     B          ; Differenz bilden
JP      P,VPOSIT   ; springe, wenn Ergebnis positiv
```

```
        NEG                 ; negiere Ergebnis
VPOSIT: NOP                 ; gemeinsame Fortsetzungsstelle
```

Weiteres Problem: Im A-Register und im H-Register soll je eine ganze Zahl in 2-Komplement-Darstellung stehen. Addiere den Inhalt des H-Registers zum Inhalt des A-Registers und schreibe das Ergebnis ins A-Register zurück. Setze das A-Register zu 0, falls die Summe positiv ist. Überläufe sollen nicht berücksichtigt werden.

Die zugehörige Formalisierung lautet:

A <– <A> + <H>
wenn <S> = 0
dann A <– 0

Als Flußdiagramm erhalten wir:

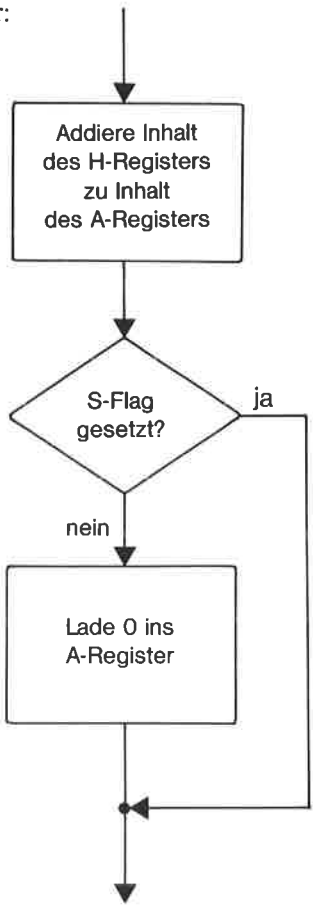


Bild 9.4. *Flußdiagramm: Abschneiden des positiven Zahlbereichs*

In diesem Fall lautet die Bedingung M (minus), die Negation von P; ein Sprung erfolgt also bei negativem Ergebnis:

```
        ADD     A,H         ; Zahlen addieren
        JP      M,VNEGAT    ; springe, wenn Ergebnis negativ
        LD      A,0         ; echt positiven Bereich abschneiden
VNEGAT: NOP                 ; gemeinsame Fortsetzungsstelle
```

Nächstes Problem: Im A-Register und im D-Register soll je eine vorzeichenlose ganze Zahl stehen. Falls der Inhalt des A-Registers kleiner als der Inhalt des D-Registers ist, soll das A-Register den Wert 0 erhalten; ansonsten soll die Differenz zwischen Inhalt des A-Registers und Inhalt des D-Registers ins A-Register gebracht werden.

Formal geschrieben (»CY« steht für das Übertrag-Flag!):

A <– <A> – <D>
wenn <CY> = 1
dann A <– 0

Im Flußdiagramm ausgedrückt:

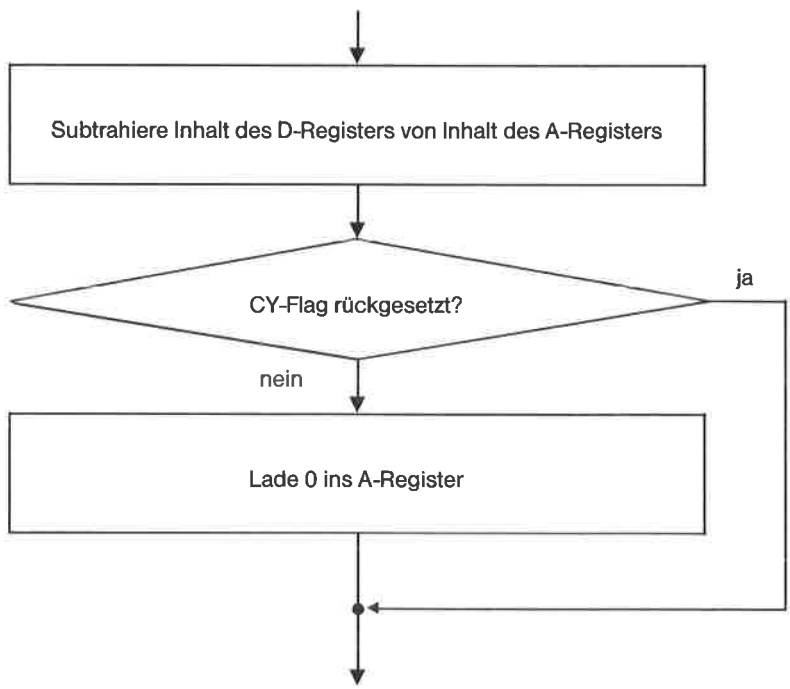


Bild 9.5. *Flußdiagramm: Abschneiden des negativen Zahlbereichs*

Die Sprungbedingung heißt hier NC (no carry):

```
        SUB     D           ; Differenz bilden
        JP      NC,NKLEIN   ; springe, wenn Inhalt des
                            ; A-Registers nicht kleiner als
                            ; Inhalt des D-Registers
        LD      A,0         ; negativen Bereich abschneiden
NKLEIN: NOP                 ; gemeinsame Fortsetzungsstelle
```

Kommen wir zu einem weiteren Problem: Im A-Register soll ein ASCII-Zeichen stehen. Falls dieses Zeichen das Ausrufezeichen ist, soll es durch ein Leerzeichen ersetzt werden.

Formal geschrieben:

wenn <A>= '!'
dann A <– ' '

Die Prüfung, ob im A-Register ein Ausrufezeichen steht, gelingt uns mit Hilfe des Befehls **CP** (compare). Der CP-Befehl wird wie ein SUB-Befehl verwendet; er setzt die Flags genau in derselben Weise wie der SUB-Befehl dies auch tun würde, verändert aber nicht den Inhalt des A-Registers.

Wir drücken den Algorithmus im Flußdiagramm aus (das »Z-Flag« ist das Null-Flag!):

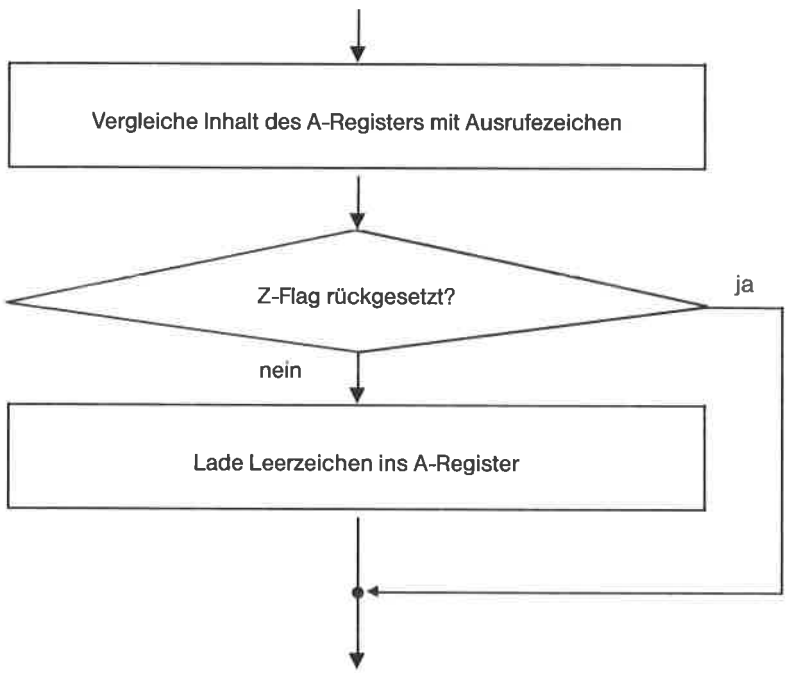


Bild 9.6. *Flußdiagramm: Ausrufezeichen durch Leerzeichen ersetzen*

Als Sprungbedingung wählen wir diesmal NZ (no zero):

```
        CP      '!'         ; Inhalt des A-Registers mit
                            ; Ausrufezeichen vergleichen
        JP      NZ,KRUFEZ   ; springe, wenn kein
                            ; Ausrufezeichen im A-Register
        LD      A,' '       ; Ausrufezeichen durch
                            ; Leerzeichen ersetzen
KRUFEZ: NOP                 ; gemeinsame Fortsetzungsstelle
```

Nächstes Problem: Im A-Register soll ein ASCII-Zeichen stehen. Ist dieses kein Leerzeichen, so wird es durch einen Punkt ersetzt.

Wir formalisieren dies:

wenn A-Register kein Leerzeichen enthält
dann A <– '.'

Der Algorithmus ist auch in folgendem Flußdiagramm zu sehen:

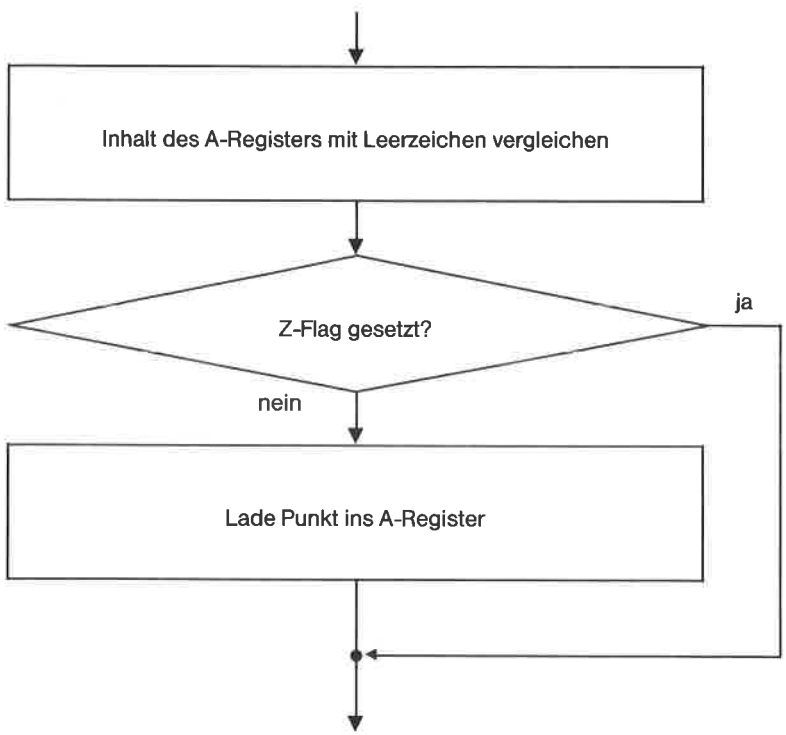


Bild 9.7. *Flußdiagramm: Nicht-Leerzeichen durch Punkt ersetzen*

Zum Prüfen, ob im A-Register ein Leerzeichen steht, verwenden wir wieder den CP-Befehl, diesmal zusammen mit der Bedingung Z (zero):

```
        CP      ' '       ; Inhalt des A-Registers
                          ; mit Leerzeichen vergleichen
        JP      Z,LEER    ; springe, wenn Leerzeichen
                          ; im A-Register
        LD      A,'.'     ; Nicht-Leerzeichen durch
                          ; Punkt ersetzen
LEER:   NOP               ; gemeinsame Fortsetzungsstelle
```

Wir kommen nun zum letzten Problem dieses Unterkapitels: Im A-Register stehe ein Buchstabe. Falls dies ein Kleinbuchstabe ist, so ersetze ihn durch den entsprechenden Großbuchstaben.

Die Formalisierung lautet:

wenn A-Register einen Kleinbuchstaben enthält
dann verwandle diesen in den entsprechenden Großbuchstaben

Da alle Großbuchstaben Codes haben, die kleiner als der Code von 'a' sind, vergleichen wir auf 'a' und springen, falls die Bedingung C (carry) erfüllt ist; ansonsten wandeln wir – wie im Kapitel »Zeichen« gezeigt – den Kleinbuchstaben in den entsprechenden Großbuchstaben um.

Wir formulieren diesen Algorithmus als Flußdiagramm:

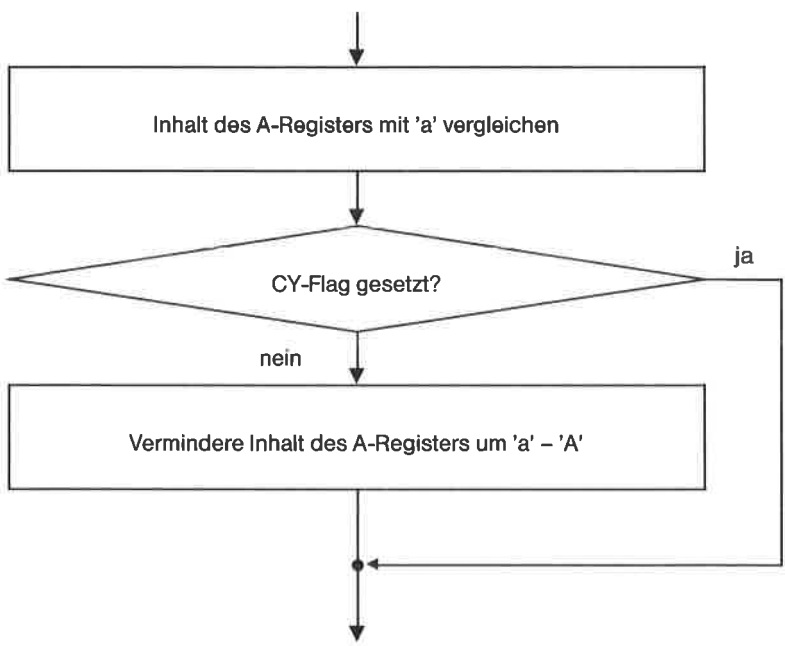


Bild 9.8. *Flußdiagramm: Umwandlung in Großbuchstaben*

Das zugehörige Programm lautet:

```
        CP      'a'         ; Inhalt des A-Registers
                            ; mit 'a' vergleichen
        JP      C,GROSS     ; springe, wenn Grossbuchstabe
                            ; im A-Register
        SUB     'a' - 'A'   ; Kleinbuchstaben in
                            ; Grossbuchstaben umwandeln
GROSS:  NOP                 ; gemeinsame Fortsetzungsstelle
```

Allen Beispielen gemeinsam ist die Technik, die in der Formalisierung angegebene Bedingung in negierter Form zum Überspringen des entsprechenden Programmteils zu verwenden.

Hier noch eine Zusammenfassung der Bedingungen für die Sprünge, die wir in den vorangegangenen Beispielen kennengelernt haben:

Z	Springe, wenn Null-Flag gesetzt
NZ	Springe, wenn Null-Flag rückgesetzt
C	Springe, wenn Übertrag-Flag gesetzt
NC	Springe, wenn Übertrag-Flag rückgesetzt
M	Springe, wenn Vorzeichen-Flag gesetzt
P	Springe, wenn Vorzeichen-Flag rückgesetzt
PE	Springe, wenn Überlauf-Flag gesetzt
PO	Springe, wenn Überlauf-Flag rückgesetzt

Merke: Keiner der Sprünge verändert die Flags!

Denke Dir zu allen Beispielen mehrere Eingabedaten aus und überprüfe die Programme damit im Debugger.

Übungen

1. Im A-Register soll ein Buchstabe stehen. Schreibe ein Programm, das Großbuchstaben in entsprechende Kleinbuchstaben umwandelt.

2. Schreibe ein Programm, das den absoluten Betrag einer ganzen Zahl in 2-Komplement-Darstellung berechnet (zur Erinnerung: der absolute Betrag *abs*(x) ist x, falls x positiv oder Null ist, ansonsten -x).

3. Für Geübte: Die Cäsar-Codierung.
   Julius Cäsar pflegte wichtige Nachrichten zu verschlüsseln, um ihren Inhalt geheimzuhalten. Er bediente sich dabei folgender Methode (übertragen auf unser deutsches Alphabet): Die großen Buchstaben des Alphabets werden fortlaufend im Kreis aufgeschrieben, so daß also auf das »Z« wieder das »A« folgt. Dann überlegt man sich, auf welchen Buchstaben das

»A« abgebildet werden soll (zum Beispiel das »C«). Alle anderen Buchstaben werden im Kreis entsprechend verschoben (aus dem »B« wird also das »D«, aus dem »C« das »E«, und so fort; aus dem »Y« wird das »A«, aus dem »Z« das »B«). Nun wird jeder Buchstabe der Nachricht durch seine Codierung ersetzt (also wird aus der Nachricht »HILFE« die verschlüsselte Nachricht »JKNHG«).
Schreibe nun ein Programm, das die Cäsar-Codierung auf einen im A-Register befindlichen Buchstaben anwendet. Der »Schlüssel« soll dabei im B-Register gespeichert sein.
Wie funktioniert die Entschlüsselung der Nachrichten? Schreibe auch hierfür ein Programm!

9.2 Zweiseitige Verzweigungen

Bei einer zweiseitigen Verzweigung wird in Abhängigkeit von einer gewissen Bedingung genau eines von zwei bestimmten Programmstücken ausgeführt. Formal ausgedrückt:

wenn Bedingung erfüllt
dann erstes Programmstück ausführen
sonst zweites Programmstück ausführen

Wir betrachten als erstes folgendes Problem: Falls im A-Register eine binär-codierte Dezimalziffer steht, so ersetze man diese durch die entsprechende ASCII-Codierung. Ansonsten schreibe man ein Leerzeichen ins A-Register.

Wir schreiben dies erst einmal formalisiert auf. Dazu überlegen wir uns, daß dic binär-codierten Dezimalziffern die Codes 00H bis 09H belegen. Ob eine Dezimalziffer im A-Register steht, stellen wir deswegen am besten durch Vergleich des Inhalts des A-Registers mit 10 fest. Zur Umwandlung in die ASCII-Darstellung bedienen wir uns der Methode aus dem Kapitel »Zeichen«. Also:

wenn <A>< 10
dann A <– <A> + '0'
sonst A <– ' '

Im Flußdiagramm ausgedrückt, Bild 9.9.

Wir wählen C (carry) als Bedingung, um einen Vorwärtssprung auszuführen, wenn im A-Register eine binär-codierte Dezimalziffer steht. Ziel des Sprunges ist das Programmstück, das die Ziffer in ASCII-Darstellung verwandelt.

Steht keine Dezimalziffer im A-Register, so führen wir statt des bedingten Sprunges das Programmstück aus, das ein Leerzeichen ins A-Register lädt. Danach müssen wir das Programm an einer gemeinsamen Stelle fortsetzen. Wir wählen dazu diejenige Stelle, die unmittelbar auf das Programmstück zur Ziffernumwandlung folgt. Dorthin gelangen wir durch einen *unbedingten*

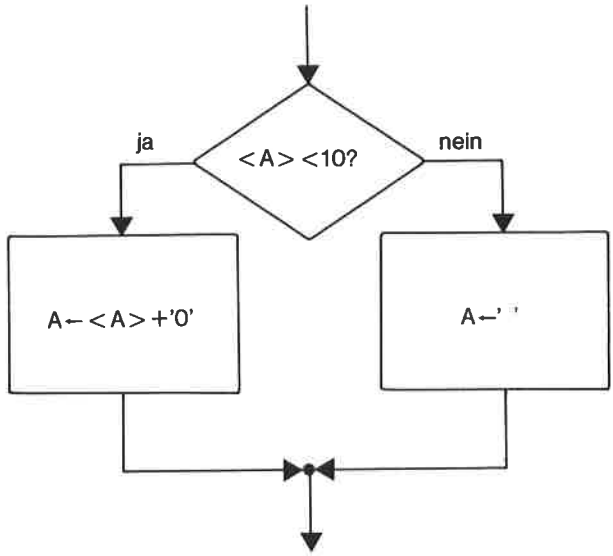


Bild 9.9. *Flußdiagramm: Beispiel einer zweiseitigen Verzweigung*

Sprung, wieder mit Hilfe des Befehls JP (jump), der dann allerdings nur einen Operanden benötigt:

```
            CP      10          ; Inhalt des A-Registers
                                ; mit 10 vergleichen
            JP      C,ZIFFER    ; springe, wenn binaer-codierte
                                ; Dezimalziffer im A-Register
            LD      A,' '       ; Bringe Leerzeichen
                                ; ins A-Register
            JP      WEITER      ; Problem geloest, weiter an
                                ; gemeinsamer Fortsetzungsstelle
ZIFFER:     ADD     A,'0'       ; binaer-codierte Dezimalziffer
                                ; in ASCII umwandeln
WEITER:     NOP                 ; gemeinsame Fortsetzungsstelle
```

Das Objekt-Programm lautet:

Adresse	Objekt-Code	Marke	Anweisung	
0000	FE 0A		CP	10
0002	DA 0A 00		JP	C,ZIFFER
0005	3E 20		LD	A,' '
0007	C3 0C 00		JP	WEITER
000A	C6 30	ZIFFER:	ADD	A,'0'
000C	00	WEITER:	NOP	

Teste das Programm mit verschiedenen Daten, zum Beispiel mit <A>=9 und <A>=10 (Grenzfälle!).

Wir hätten ohne weiteres die Reihenfolge der beiden Programmstücke vertauschen können, was auf folgendes Programm führen würde:

```
        CP      10          ; Inhalt des A-Registers
                            ; mit 10 vergleichen
        JP      NC,KZIFF    ; springe, wenn keine binaer-
                            ; codierte Dezimalziffer
                            ; im A-Register
        ADD     A,'0'       ; binaer-codierte Dezimalziffer
                            ; in ASCII umwandeln
        JP      WEITER      ; Problem geloest, weiter an
                            ; gemeinsamer Fortsetzungsstelle
KZIFF:  LD      A,' '       ; Bringe Leerzeichen
                            ; ins A-Register
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

Teste auch dieses Programm mit vernünftigen Daten!

Merke: Eine zweiseitige Verzweigung kann stets auf zwei Weisen programmiert werden, jeweils mit Hilfe eines bedingten Sprungs auf die Behandlung des einen Falls und eines unbedingten Sprungs am Ende der Behandlung des anderen Falls!

Wir betrachten als nächstes folgendes Problem: Im B-Register stehe eine ganze Zahl in 2-Komplement-Darstellung. Wenn der Inhalt des A-Registers gleich 1 ist, soll der Inhalt des B-Registers in das A-Register übertragen werden. Ansonsten soll der Inhalt des B-Registers mit umgekehrtem Vorzeichen ins A-Register gebracht werden (das A-Register wirkt damit durch seinen ursprünglichen Inhalt ähnlich wie ein Flag).

Die Formalisierung lautet:

wenn <A> = 1
dann A <– <B>
sonst A <– 0 – <B>

Auf ein Flußdiagramm verzichten wir diesmal; wir werden im folgenden nur noch dann Flußdiagramme angeben, wenn neue oder komplizierte Ablaufstrukturen behandelt werden. Auch mit Kommentaren werden wir in Zukunft etwas sparsamer sein, insbesondere dort, wo die Befehle für sich selbst sprechen.

Wir können als Sprungbedingung Z oder NZ wählen. Entscheiden wir uns für Z, so lautet unser Programm:

```
        CP      1
        JP      Z,EINS      ; springe, wenn Inhalt des
                            ; A-Registers gleich 1 ist
        LD      A,B
        NEG
        JP      WEITER      ; Problem geloest, weiter an
                            ; gemeinsamer Fortsetzungsstelle
EINS:   LD      A,B
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

Als Objekt-Code erhalten wir:

Adresse	Objekt-Code	Marke	Anweisung	
0000	FE 01		CP	1
0002	CA 0B 00		JP	Z,EINS
0005	78		LD	A,B
0006	ED 44		NEG	
0008	C3 0C 00		JP	WEITER
000B	78	EINS:	LD	A,B
000C	00	WEITER:	NOP	

Wir erinnern uns nun daran, daß der LD-Befehl die Flags nicht verändert. Wir könnten also auch zuerst den Inhalt des A-Registers auf 1 testen, dann den Inhalt des B-Registers ins A-Register umladen, und als letztes den Inhalt des A-Registers negieren, falls der ursprüngliche Inhalt ungleich 1 war:

```
        CP      1
        LD      A,B
        JP      Z,EINS      ; springe, wenn Inhalt des
                            ; A-Registers gleich 1 war
        NEG
EINS:   NOP                 ; gemeinsame Fortsetzungsstelle
```

Der zugehörige Objekt-Code lautet:

Adresse	Objekt-Code	Marke	Anweisung	
0000	FE 01		CP	1
0002	78		LD	A,B
0003	CA 08 00		JP	Z,EINS
0006	ED 44		NEG	
0008	00	EINS:	NOP	

Wir haben durch Reorganisation des Programms zwei Befehle (deren Objekt-Code in diesem Fall 4 Bytes belegt) eingespart. Der Blick für solche Optimierungen zeichnet den versierten Programmierer aus; der Anfänger sollte auf keinen Fall versuchen, Programmentwicklung und Optimierung in einem Schritt zu erledigen!

Merke: Prüfe zweiseitige Verzweigungen stets darauf, ob sie auch als einseitige Verzweigung ausgeführt werden können!
An dieser Stelle soll etwas genauer auf die Optimierung von Programmen eingegangen werden. Wir unterscheiden zwei Optimierungsziele:

- Laufzeitoptimierung
- Speicheroptimierung

Bei der Laufzeitoptimierung versucht man, den Erwartungswert für die zum Durchlauf des Programms nötige Rechenzeit möglichst niedrig zu halten. Dazu bedient man sich der Kenntnis der Laufzeit der einzelnen Befehle, die für den Z80 in Takt-Zyklen angegeben werden (um unabhängig von der Taktfrequenz rechnen zu können). Nachfolgend einige Beispiele:

LD	register,register	4 Takt-Zyklen
LD	register,konstante	7 Takt-Zyklen
ADD	A,register	4 Takt-Zyklen
ADD	A,konstante	7 Takt-Zyklen
CP	konstante	7 Takt-Zyklen
JP	adresse	10 Takt-Zyklen
JP	bedingung,adresse	10 Takt-Zyklen

Bei der Speicheroptimierung ist man bestrebt, den Speicherbedarf für den Objekt-Code und den durchschnittlichen oder maximalen Speicherbedarf der Daten (Schwankungen werden durch dynamische Datenstrukturen verursacht) zu minimieren. Beispiele für den Speicherbedarf von Befehlen sind:

LD	register,register	1 Byte
LD	register,konstante	2 Bytes
ADD	A,register	1 Byte
ADD	A,konstante	2 Bytes
CP	konstante	2 Bytes
JP	adresse	3 Bytes
JP	bedingung,adresse	3 Bytes

Die Durchlaufzeiten und der Speicherbedarf für den Objekt-Code der einzelnen Befehle können dem Anhang B entnommen werden.

In manchen Fällen lassen sich Laufzeitoptimierung und Speicheroptimierung verbinden; meist jedoch widersprechen sie sich, und man wird sich für eines der beiden Ziele entscheiden (oder einen Kompromiß eingehen).

Zur Unterstützung von Speicheroptimierungen existiert noch ein weiterer unbedingter Sprungbefehl, nämlich **JR** (jump relative). Dieser hat als Argument im Assemblerprogramm zwar eine Absolutadresse, der Assembler setzt diese aber in eine Relativadresse um, die sich auf die Anfangsadresse des auf den JR-Befehl folgenden Befehls bezieht und von dort aus Vorwärtssprünge um maximal 127 Bytes, Rückwärtssprünge um maximal 128 Bytes erlaubt; dies liegt daran, daß die Relativadresse im Objekt-Code als 8-Bit Größe im 2-Komplement gespeichert ist. Da der Objekt-Code des JR-Befehls 2 Bytes belegt, ist dieser speicherökonomischer als der JP-Befehl. Ein weiterer Vorteil ist, daß Programmstücke, die ausschließlich mit relativen Sprüngen arbeiten, im Speicher frei verschiebbar (engl. relocatable) sind (siehe Kapitel »Verschiebbare Programme«). Die Durchlaufzeit des unbedingten JR-Befehls beträgt 12 Takt-Zyklen und liegt damit über der eines JP-Befehls. Man sieht hier gut, wie unsere beiden Optimierungsziele kollidieren.

Wir wollen nun das erste Beispielprogramm dieses Unterkapitels nochmals hervorholen und durch einen unbedingten relativen Sprung speicheroptimieren (der Objekt-Code wird dabei um ein Byte kürzer):

```
            CP      10          ; Inhalt des A-Registers
                                ; mit 10 vergleichen
            JP      C,ZIFFER    ; springe, wenn binaer-codierte
                                ; Dezimalziffer im A-Register
            LD      A,' '       ; Bringe Leerzeichen
                                ; ins A-Register
            JR      WEITER      ; Problem geloest, weiter an
                                ; gemeinsamer Fortsetzungsstelle
ZIFFER:     ADD     A,'0'       ; binaer-codierte Dezimalziffer
                                ; in ASCII umwandeln
WEITER:     NOP                 ; gemeinsame Fortsetzungsstelle
```

Wir sehen uns gleich das Objekt-Programm an, um den Effekt zu kontrollieren:

Adresse	Objekt-Code	Marke	Anweisung	
0000	FE 0A		CP	10
0002	DA 09 00		JP	C,ZIFFER
0005	3E 20		LD	A,' '
0007	18 02		JR	WEITER
0009	C6 30	ZIFFER:	ADD	A,'0'
000B	00	WEITER:	NOP	

Außer dem unbedingten relativen Sprung gibt es noch einige bedingte relative Sprungbefehle, und zwar zu folgenden Bedingungen:

Z	Springe, wenn Null-Flag gesetzt
NZ	Springe, wenn Null-Flag rückgesetzt

C	Springe, wenn Übertrag-Flag gesetzt
NC	Springe, wenn Übertrag-Flag rückgesetzt

Ein bedingter relativer Sprung wird so notiert wie ein bedingter absoluter Sprung, nur eben mit dem Befehlsnamen »JR« statt »JP«. Der Objekt-Code eines bedingten relativen Sprungs belegt 2 Bytes; die bedingten relativen Sprünge sind damit gleichfalls speicherökonomischer als die bedingten absoluten Sprünge. Die Durchlaufzeit eines bedingten relativen Sprungs hängt davon ab, ob die Bedingung erfüllt ist. Bei erfüllter Bedingung benötigt der Befehl 12 Takt-Zyklen, bei nicht erfüllter Bedingung dagegen nur 7 Takt-Zyklen. Dieses Phänomen erlaubt nun sogar eine Laufzeitoptimierung durch bedingte relative Sprünge.

Wir studieren das an folgendem Beispiel: Im A-Register soll eine vorzeichenlose ganze Zahl stehen. Wenn diese größer als 26 ist, soll 0 ins B-Register geladen werden. Ansonsten soll 1 ins B-Register gebracht werden.

Die Formalisierung des Problems lautet:

wenn	<A> > 26
dann	B <— 0
sonst	B <— 1

Es folgen vier mögliche Varianten des Programms (jeweils mit dem zugehörigen Objekt-Code):

```
VARI1:  LD      B,0         ; vorsorglich den Wert 0 laden
                            ; wir erwarten, dass der Inhalt
                            ; des A-Registers groesser
                            ; als 26 ist
        CP      27
        JP      NC,GROESS   ; springe, wenn Inhalt des
                            ; A-Registers groesser als 26
        LD      B,1
GROESS: NOP
```

Adresse	Objekt-Code	Marke	Anweisung	
0000	06 00	VARI1:	LD	B,0
0002	FE 1B		CP	27
0004	D2 09 00		JP	NC,GROESS
0007	06 01		LD	B,1
0009	00	GROESS:	NOP	

```
VARI2:  LD      B,1         ; vorsorglich den Wert 1 laden
                            ; wir erwarten, dass der Inhalt
                            ; des A-Registers kleiner oder
                            ; gleich 26 ist
```

```
            CP      27
            JP      C,KLEIN     ; springe, wenn Inhalt des
                                ; A-Registers kleiner gleich 26
            LD      B,0
KLEIN:      NOP
```

Adresse	Objekt-Code	Marke	Anweisung	
0000	06 01	VARI2:	LD	B,1
0002	FE 1B		CP	27
0004	DA 09 00		JP	C,KLEIN
0007	06 00		LD	B,0
0009	00	KLEIN:	NOP	

```
VARI3:      LD      B,0         ; vorsorglich den Wert 0 laden
                                ; wir erwarten, dass der Inhalt
                                ; des A-Registers groesser
                                ; als 26 ist
            CP      27
            JR      NC,GROESS   ; springe, wenn Inhalt des
                                ; A-Registers groesser als 26
            LD      B,1
GROESS:     NOP
```

Adresse	Objekt-Code	Marke	Anweisung	
0000	06 00	VARI3:	LD	B,0
0002	FE 1B		CP	27
0004	30 02		JR	NC,GROESS
0006	06 01		LD	B,1
0008	00	GROESS:	NOP	

```
VARI4:      LD      B,1         ; vorsorglich den Wert 1 laden
                                ; wir erwarten, dass der Inhalt
                                ; des A-Registers kleiner oder
                                ; gleich 26 ist
            CP      27
            JR      C,KLEIN     ; springe, wenn Inhalt des
                                ; A-Registers kleiner gleich 26
            LD      B,0
KLEIN:      NOP
```

Adresse	Objekt-Code	Marke	Anweisung	
0000	06 01	VARI4:	LD	B,1
0002	FE 1B		CP	27
0004	38 02		JR	C,KLEIN
0006	06 00		LD	B,0
0008	00	KLEIN:	NOP	

Wir bezeichnen mit p die Wahrscheinlichkeit, daß im A-Register ein Wert kleiner als 27 steht. Die durchschnittlichen Laufzeiten t(p) unserer vier Programme (den NOP-Befehl wollen wir als nicht zum Programm gehörig ansehen) hängen von p ab.

Zuerst fertigen wir eine Tabelle der Laufzeiten der einzelnen Befehle an:

LD	B,0	7 Takt-Zyklen
LD	B,1	7 Takt-Zyklen
CP	27	7 Takt-Zyklen
JP	C,KLEIN	10 Takt-Zyklen
JP	NC,GROESS	10 Takt-Zyklen
JR	C,KLEIN	12 Takt-Zyklen, falls <CY> = 1
		7 Takt-Zyklen, falls <CY> = 0
JR	NC,GROESS	12 Takt-Zyklen, falls <CY> = 0
		7 Takt-Zyklen, falls <CY> = 1

Die durchschnittliche Laufzeit der ersten Variante berechnet sich wie folgt: Die ersten drei Befehle besitzen eine konstante Durchlaufzeit und werden auf jeden Fall durchgeführt. Der vierte Befehl wird nur durchgeführt, falls der Inhalt des A-Registers kleiner als 27 ist, also mit der Wahrscheinlichkeit p. Dies ergibt

$$t_1(p) = 7 + 7 + 10 + 7 * p = 24 + 7 * p.$$

Auf dieselbe Weise erhalten wir als durchschnittliche Laufzeit der zweiten Variante

$$t_2(p) = 7 + 7 + 10 + 7 * (1-p) = 31 - 7 * p.$$

Bei der dritten Variante besitzen die ersten beiden Befehle eine konstante Durchlaufzeit und werden stets ausgeführt. Der relative Sprung verzweigt mit der Wahrscheinlichkeit (1—p) zur symbolischen Adresse GROESS (12 Takt-Zyklen); mit der Wahrscheinlichkeit p erfolgt kein Sprung (7 Takt-Zyklen), es wird jedoch der vierte Befehl ausgeführt. Wir erhalten so

$$t_3(p) = 7 + 7 + 12 * (1-p) + (7 + 7) * p = 26 + 2 * p.$$

Für die vierte Variante ergibt sich analog

$t_4(p) = 7 + 7 + 12 * p + (7 + 7) * (1-p) = 28 - 2 * p.$

Je nach Größe von p wählen wir jetzt die Variante mit der minimalen durchschnittlichen Laufzeit:

Variante 1, falls 0.0 <= p <= 0.4
Variante 3, falls 0.4 <= p <= 0.5
Variante 4, falls 0.5 <= p <= 0.6
Variante 2, falls 0.6 <= p <= 1.0

Die Überlegungen gelten natürlich nur, wenn durch die Reorganisation die Durchlaufzeit des restlichen Programms nicht verändert wird. Die Methode läßt sich in entsprechender Form auch für zweiseitige Verzweigungen (und andere Kontrollstrukturen, die wir noch kennenlernen werden) anwenden.

Nun noch je ein Beispiel für einen bedingten relativen Sprung mit der Bedingung »Z« beziehungsweise »NZ«:

Wir betrachten folgendes Problem: Wenn im A-Register ein Leerzeichen steht, soll das A-Register den Wert 0 erhalten, sonst soll der Inhalt des B-Registers negiert ins A-Register gebracht werden.

Formal aufgeschrieben:

wenn <A> = ' '
dann A <- 0
sonst A <- 0 - <B>

Das Programm lautet dann beispielsweise:

```
        CP      ' '
        JR      Z,LEER      ; springe, wenn Leerzeichen
                            ; im A-Register
        LD      A,B
        NEG
        JR      WEITER      ; Problem geloest, weiter an
                            ; gemeinsamer Fortsetzungsstelle
LEER:   LD      A,0
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

Als letztes Problem untersuchen wir folgendes: Wenn im A-Register der Buchstabe 'R' steht, soll das A-Register mit dem Inhalt des D-Registers geladen werden, ansonsten mit dem Inhalt der Speicherzelle mit der Adresse 139FH.

Die formale Schreibweise lautet also:

wenn <A> = 'R'
dann A <– <D>
sonst A <– <(139FH)>

Das Programm lautet dann:

```
        CP      'R'
        JR      NZ,NICHTR   ; springe, wenn kein 'R'
                            ; im A-Register
        LD      A,D
        JR      WEITER      ; Problem geloest, weiter an
                            ; gemeinsamer Fortsetzungsstelle
NICHTR: LD      A,(139FH)
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

In beiden Fällen könnten wir noch eine Laufzeitoptimierung versuchen.

Wir werden die relativen Sprünge wegen ihrer Einschränkungen (kurze Sprungdistanz und nur bestimmte Flags testbar) in den weiteren Beispielen nicht mehr verwenden (es sei denn, um optimierte Programme vorzustellen). Versuche ab und zu einmal, relative Sprünge zur Laufzeit- oder Speicheroptimierung der im weiteren Verlauf des Buches noch folgenden Programme zu verwenden!

Übungen

1. Im A-Register stehe ein Buchstabe. Schreibe ein Programm, das Groß- und Kleinbuchstaben entsprechend vertauscht.

2. Schreibe ein Programm, das eine binär codierte Hex-Ziffer in die entsprechende ASCII-Darstellung (Dezimalziffer oder Großbuchstabe) überführt.

3. Schreibe Programme, die eine ASCII-codierte Hex-Ziffer in ihre Binärdarstellung umwandeln. Verwende dazu alternativ
   - eine zweiseitige Verzweigung
   - eine einseitige Verzweigung

   Analysiere die Unterschiede zwischen den beiden Versionen.

9.3 Verzweigungsketten

Bei vielen Problemen werden wir nicht mit einer zweiseitigen Verzweigung auskommen, weil mehr als zwei Fälle zu berücksichtigen sind. Wir können dann meistens durch Abprüfen einer Bedingung jeweils einen Teilfall isolieren und müssen mit Hilfe weiterer Bedingungen die übrigen Teilfälle erledigen. Dies führt zu einer Verzweigungskette, die formal so beschrieben werden kann:

```
wenn    Bedingung 1 erfüllt
dann    Programmstück 1 ausführen
sonst   wenn Bedingung 2 erfüllt
        dann    Programmstück 2 ausführen
        sonst   .
                .
                .
                wenn    Bedingung n erfüllt
                dann    Programmstück n ausführen
                sonst   Programmstück n+1 ausführen
```

Wir sehen uns dazu einige Beispiele an:

Betrachte folgendes Problem: Im A-Register stehe ein Zeichen. Falls dieses keine ASCII-codierte Dezimalziffer ist, soll es durch ein Fragezeichen ersetzt werden.

Wir haben hier drei Fälle zu unterscheiden, nämlich

1) im A-Register steht eine ASCII-codierte Dezimalziffer,
2) im A-Register steht ein Zeichen mit einem ASCII-Code kleiner als '0',
3) im A-Register steht ein Zeichen mit einem ASCII-Code größer als '9'.

Als Verzweigungskette formuliert lautet das Problem:

```
wenn    <A>< '0'
dann    A <– '?'
sonst   wenn    <A>> '9'
        dann    A <– '?'
        sonst   tue nichts
```

Wir organisieren das Programm so, daß wir möglichst wenig Sprünge benötigen. Da wir zwei Bedingungen prüfen, müssen mindestens zwei Sprünge verwendet werden. Statt der Bedingung <A>> '9' wählen wir deren Negation <A>< 1+'9':

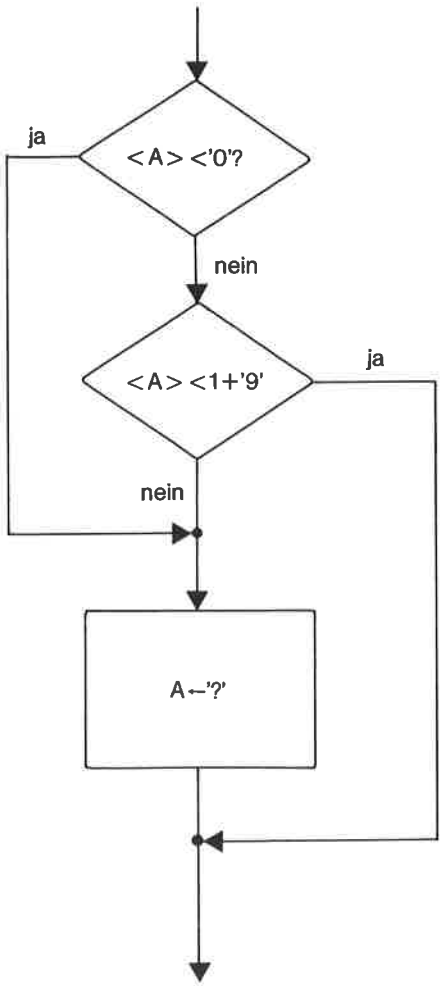


Bild 9.10. *Flußdiagramm: Beispiel einer Verzweigungskette*

Das Programm lautet also:

```
CP      '0'
JP      C,FRAGEZ    ; wenn Inhalt des A-Registers
                    ; kleiner als ASCII-Null,
                    ; springe und ersetze Zeichen
                    ; durch Fragezeichen
CP      1+'9'
JP      C,WEITER    ; wenn Inhalt des A-Registers
```

```
                                        ; nicht groesser als ASCII-Neun,
                                        ; Dezimalziffer im A-Register,
                                        ; Problem geloest, weiter an
                                        ; gemeinsamer Fortsetzungsstelle
FRAGEZ:   LD        A,'?'
WEITER:   NOP                           ; gemeinsame Fortsetzungsstelle
```

Nächstes Problem: Im A-Register stehe eine ganze Zahl in 2-Komplement-Darstellung. Ersetze diese durch ihr Signum.

Wir erinnern uns an die Definition der Funktion Signum:

$$\text{Signum}(x) = \begin{cases} -1, & \text{falls } x < 0 \\ 0, & \text{falls } x = 0 \\ 1, & \text{falls } x > 0 \end{cases}$$

Die zugehörige Verzweigungskette lautet:

wenn <A>< 0
dann A <– –1
sonst **wenn** <A>= 0
dann tue nichts
sonst A <– 1

Wir versuchen wieder mit einer minimalen Anzahl von Sprüngen auszukommen. Um die drei Fälle zu unterscheiden, vergleichen wir den Inhalt des A-Registers mit 0:

```
          CP        0
          JP        M,NEGAT     ; Operand negativ
          JP        Z,WEITER    ; Operand Null, nichts zu tun
          LD        A,1
          JP        WEITER      ; Problem geloest, weiter an
                                ; gemeinsamer Fortsetzungsstelle
NEGAT:    LD        A,-1
WEITER:   NOP                   ; gemeinsame Fortsetzungsstelle
```

Der zugehörige Objekt-Code lautet:

Adresse	Objekt-Code	Marke	Anweisung	
0000	FE 00		CP	0
0002	FA 0D 00		JP	M,NEGAT
0005	CA 0F 00		JP	Z,WEITER
0008	3E 01		LD	A,1

```
000A    C3 0F 00            JP      WEITER
000D    3E FF       NEGAT:  LD      A,-1
000F    00          WEITER: NOP
```

Wir reorganisieren jetzt das Problem, um einen Sprung einsparen zu können:

wenn <A> = 0
dann tue nichts
sonst **wenn** <A> < 0
dann A <- -1
sonst A <- 1

Das zugehörige Programm lautet:

```
        CP      0
        JP      Z,WEITER    ; Operand Null, nichts zu tun
        LD      A,-1        ; wir nehmen an, daß Operand
                            ; negativ war
        JP      M,WEITER    ; Operand war negativ,
                            ; Problem geloest
        LD      A,1
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

Als Objekt-Code erhalten wir nun:

Adresse	Objekt-Code	Marke	Anweisung	
0000	FE 00		CP	0
0002	CA 0C 00		JP	Z,WEITER
0005	3E FF		LD	A,-1
0007	FA 0C 00		JP	M,WEITER
000A	3E 01		LD	A,1
000C	00	WEITER:	NOP	

Weiteres Problem: Das A-Register enthält ein Zeichen. Falls dieses keine ASCII-codierte Hex-Ziffer darstellt, soll es durch ein Leerzeichen ersetzt werden (eine Hex-Ziffer ist entweder eine Dezimal-Ziffer oder einer der Großbuchstaben A, B, C, D, E, F; natürlich könnten wir auch die entsprechenden Kleinbuchstaben zulassen).

Wir formulieren eine Verzweigungskette:

```
wenn    <A>< '0'
dann    A <– ' '
sonst   wenn    <A><= '9'
        dann    tue nichts
        sonst   wenn    <A>< 'A'
                dann    A <– ' '
                sonst   wenn    <A><= 'F'
                        dann    tue nichts
                        sonst   A <– ' '
```

Wir zeigen den Algorithmus im Flußdiagramm (Bild 9.11.).

Wir setzen dies schematisch in ein Programm um (beachte dabei die Realisierung der Relation »kleiner gleich«!):

```
        CP      '0'
        JP      C,LEERZ     ; Keine Hex-Ziffer, durch
                            ; Leerzeichen ersetzen
        CP      1+'9'
        JP      C,WEITER    ; Dezimalziffer, nichts zu tun
        CP      'A'
        JP      C,LEERZ     ; Keine Hex-Ziffer, durch
                            ; Leerzeichen ersetzen
        CP      1+'F'
        JP      C,WEITER    ; Hex-Ziffer (A-F), nichts zu tun
LEERZ:  LD      A,' '       ; Zeichen durch
                            ; Leerzeichen ersetzen
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

Wir kommen nun zum letzten und größten Problem dieses Unterkapitels: Wir wollen einen Codierer realisieren. Im A-Register stehe ein Zeichen; diesem soll ein Funktionscode zugeordnet werden nach folgender Vorschrift:

sichtbares Zeichen	–>	1
BS	–>	2
HT	–>	3
LF	–>	4
FF	–>	5
CR	–>	6
anderes Steuerzeichen	–>	7
sonstiges Zeichen	–>	–1

Der Funktionscode soll ins B-Register geschrieben werden.

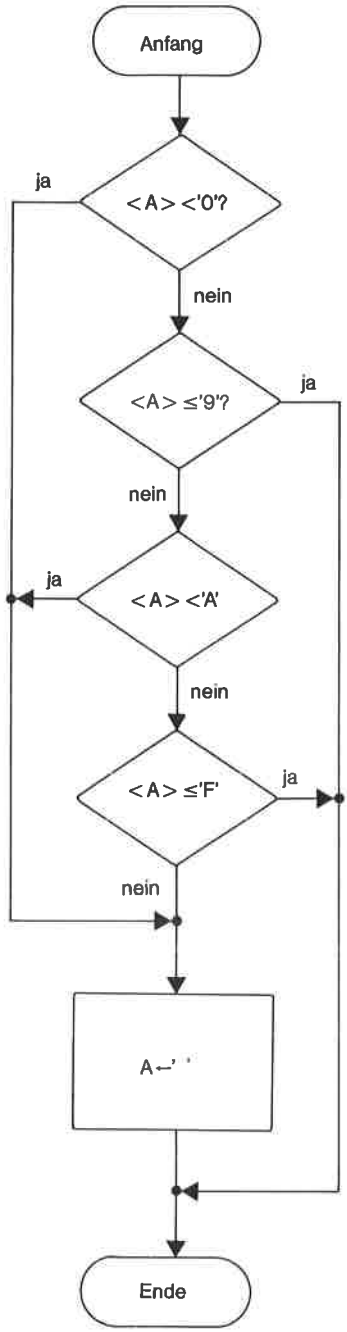


Bild 9.11. *Flußdiagramm: Zeichen als Hex-Ziffer identifizieren*

Wir organisieren das Programm so, daß bei jeder Entscheidung ein bestimmtes Zeichen oder ein Menge bestimmter Zeichen isoliert und codiert wird. Dabei sollen große Mengen möglichst bald erledigt werden, da sie mit hoher Wahrscheinlichkeit das Zeichen enthalten. Die Verzweigungskette lautet:

```
wenn    <A>= DEL
dann    B <– 7
sonst   wenn    <A> > DEL
        dann    B <– –1
        sonst   wenn    <A> >= SP
                dann    B <– 1
                sonst   wenn    <A>= BS
                        dann    B <– 2
                        sonst   wenn    <A>= HT
                                dann    B <– 3
                                sonst   wenn    <A>= LF
                                        dann    B <– 4
                                        sonst   wenn    <A>= FF
                                                dann    B <– 5
                                                sonst   wenn    <A>= CR
                                                        dann    B <– 6
                                                        sonst   B <– 7
```

Wir demonstrieren den Algorithmus auch noch im Flußdiagramm (Bild 9.12.).

In dem entsprechenden Programm verwenden wir die Assembler-Pseudo-Operation **EQU** (equate). Mit einer EQU-Operation wird einem Namen ein fester Wert zugewiesen (benannte Konstante). Wir setzen alle verwendeten EQU-Operationen an den Anfang des Programms, was sehr zur Klarheit beiträgt:

```
BS          EQU         08H
HT          EQU         09H
LF          EQU         0AH
FF          EQU         0CH
CR          EQU         0DH
SP          EQU         20H
DEL         EQU         7FH

            CP          DEL
            JP          NZ,KDEL ; kein DEL gefunden
            LD          B,7
            JP          WEITER      ; Problem geloest
KDEL:       JP          C,KSONST    ; kein sonstiges Zeichen
                                    ; gefunden
```

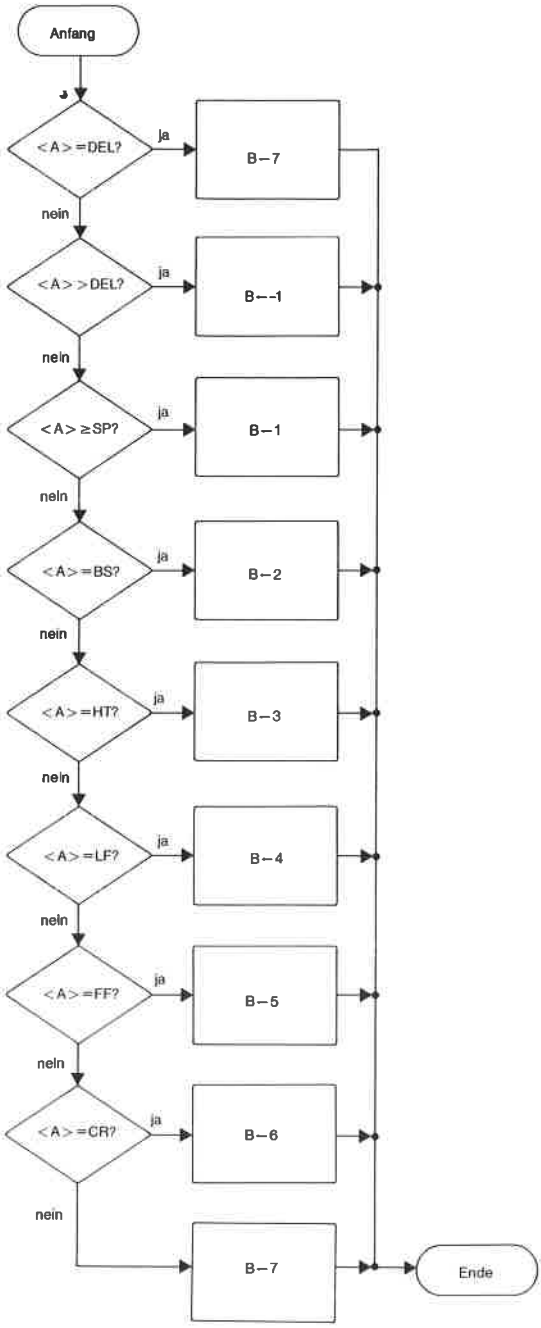


Bild 9.12. *Flußdiagramm: Codierer*

```
            LD      B,-1
            JP      WEITER      ; Problem geloest
KSONST:     CP      SP
            JP      C,KSICHT    ; kein sichtbares Zeichen
                                ; gefunden
            LD      B,1
            JP      WEITER      ; Problem geloest
KSICHT:     CP      BS
            JP      NZ,KBS      ; kein BS gefunden
            LD      B,2
            JP      WEITER      ; Problem geloest
KBS:        CP      HT
            JP      NZ,KHT      ; kein HT gefunden
            LD      B,3
            JP      WEITER      ; Problem geloest
KHT:        CP      LF
            JP      NZ,KLF      ; kein LF gefunden
            LD      B,4
            JP      WEITER      ; Problem geloest
KLF:        CP      FF
            JP      NZ,KFF      ; kein FF gefunden
            LD      B,5
            JP      WEITER      ; Problem geloest
KFF:        CP      CR
            JP      NZ,KCR      ; kein CR gefunden
            LD      B,6
            JP      WEITER      ; Problem geloest
KCR:        LD      B,7         ; anderes Steuerzeichen gefunden
WEITER:     NOP                 ; gemeinsame Fortsetzungsstelle
```

Als Objekt-Code erhalten wir dabei:

```
Adresse   Objekt-Code   Marke   Anweisung
          0008          BS      EQU     08H
          0009          HT      EQU     09H
          000A          LF      EQU     0AH
          000C          FF      EQU     0CH
          000D          CR      EQU     0DH
          0020          SP      EQU     20H
          007F          DEL     EQU     7FH

0000      FE 7F                 CP      DEL
0002      C2 0A 00              JP      NZ,KDEL
```

```
0005    06 07          LD      B,7
0007    C3 50 00       JP      WEITER
000A    DA 12 00 KDEL: JP      C,KSONST
000D    06 FF          LD      B,-1
000F    C3 50 00       JP      WEITER
0012    FE 20    KSONST: CP    SP
0014    DA 1C 00       JP      C,KSICHT
0017    06 01          LD      B,1
0019    C3 50 00       JP      WEITER
001C    FE 08    KSICHT: CP    BS
001E    C2 26 00       JP      NZ,KBS
0021    06 02          LD      B,2
0023    C3 50 00       JP      WEITER
0026    FE 09    KBS:  CP      HT
0028    C2 30 00       JP      NZ,KHT
002B    06 03          LD      B,3
002D    C3 50 00       JP      WEITER
0030    FE 0A    KHT:  CP      LF
0032    C2 3A 00       JP      NZ,KLF
0035    06 04          LD      B,4
0037    C3 50 00       JP      WEITER
003A    FE 0C    KLF:  CP      FF
003C    C2 44 00       JP      NZ,KFF
003F    06 05          LD      B,5
0041    C3 50 00       JP      WEITER
0044    FE 0D    KFF:  CP      CR
0046    C2 4E 00       JP      NZ,KCR
0049    06 06          LD      B,6
004B    C3 50 00       JP      WEITER
004E    06 07    KCR:  LD      B,7
0050    00       WEITER: NOP
```

Wir machen uns hier wiederum die Tatsache zunutze, daß LD-Befehle wegen ihren geringen Seiteneffekte recht freizügig im Programm verschoben werden können, und erhalten folgende optimierte Variante des Programms:

```
BS      EQU     08H
HT      EQU     09H
LF      EQU     0AH
FF      EQU     0CH
CR      EQU     0DH
SP      EQU     20H
DEL     EQU     7FH
```

```
        LD      B,7
        CP      DEL
        JP      Z,WEITER    ; DEL gefunden
        LD      B,-1
        JP      NC,WEITER   ; sonstiges Zeichen gefunden
        LD      B,1
        CP      SP
        JP      NC,WEITER   ; sichtbares Zeichen gefunden
        LD      B,2
        CP      BS
        JP      Z,WEITER    ; BS gefunden
        LD      B,3
        CP      HT
        JP      Z,WEITER    ; HT gefunden
        LD      B,4
        CP      LF
        JP      Z,WEITER    ; LF gefunden
        LD      B,5
        CP      FF
        JP      Z,WEITER    ; FF gefunden
        LD      B,6
        CP      CR
        JP      Z,WEITER    ; CR gefunden
        LD      B,7         ; anderes Steuerzeichen gefunden
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

Als Objekt-Code ergibt sich nun:

Adresse	Objekt-Code	Marke	Anweisung	
	0008	BS	EQU	08H
	0009	HT	EQU	09H
	000A	LF	EQU	0AH
	000C	FF	EQU	0CH
	000D	CR	EQU	0DH
	0020	SP	EQU	20H
	007F	DEL	EQU	7FH
0000	0607		LD	B,7
0002	FE 7F		CP	DEL
0004	CA 38 00		JP	Z,WEITER
0007	06 FF		LD	B,−1
0009	D2 38 00		JP	NC,WEITER
000C	06 01		LD	B,1

```
000E    FE 20              CP      SP
0010    D2 38 00           JP      NC,WEITER
0013    06 02              LD      B,2
0015    FE 08              CP      BS
0017    CA 38 00           JP      Z,WEITER
001A    06 03              LD      B,3
001C    FE 09              CP      HT
001E    CA 38 00           JP      Z,WEITER
0021    06 04              LD      B,4
0023    FE 0A              CP      LF
0025    CA 38 00           JP      Z,WEITER
0028    06 05              LD      B,5
002A    FE 0C              CP      FF
002C    CA 38 00           JP      Z,WEITER
002F    06 06              LD      B,6
0031    FE 0D              CP      CR
0033    CA 38 00           JP      Z,WEITER
0036    06 07              LD      B,7
0038    00         WEITER: NOP
```

Übungen

1. Optimiere das erste Beispielprogramm von Kapitel 9.3. Vergleiche die durchschnittlichen Laufzeiten beider Programme unter der Annahme, daß alle Zeichen gleich häufig sind.

2. Schreibe ein Programm, das Kleinbuchstaben ('a' bis 'z') in die entsprechenden Großbuchstaben umwandelt.

3. Schreibe ein Programm, das mit einer im B-Register stehenden Binärzahl folgendermaßen verfährt: Falls die Zahl einer Hexadezimalziffer entspricht (00H bis 0FH), soll diese in ihr ASCII-Äquivalent gewandelt werden; darüber hinaus soll 0 ins E-Register geschrieben werden. Andernfalls soll −1 ins E-Register geschrieben werden; das B-Register wird mit '?' gefüllt.

4. Erweitere das Programm aus Aufgabe 3 so, daß auch ASCII-codierte Hexadezimalziffern erkannt werden (sowohl Groß- als auch Kleinbuchstaben sollen dabei gültig sein); diese sollen dann in ihr binäres Pendant umgewandelt werden. Das E-Register erhält in diesem Fall den Wert 1.

9.4 Verzweigungskaskaden

Mitunter ist es bei komplizierten Problemen nicht möglich, die Teilfälle sukzessive durch Abprüfen jeweils einer Bedingung zu isolieren. Wir zerlegen dann die Menge der Fälle durch Prüfen einer Bedingung in zwei kleinere Teilmengen und behandeln diese anschließend getrennt weiter. Das allgemeine Resultat dieses Vorgangs ist eine Verzweigungskaskade. Wir beschreiben eine vollständige Verzweigungskaskade mit drei Stufen:

```
wenn    Bedingung erfüllt
dann    wenn    Bedingung 1 erfüllt
        dann    wenn    Bedingung 1.1 erfüllt
                dann    Programmstück 1.1.1 ausführen
                sonst   Programmstück 1.1.2 ausführen
        sonst   wenn    Bedingung 1.2 erfüllt
                dann    Programmstück 1.2.1 ausführen
                sonst   Programmstück 1.2.2 ausführen
sonst   wenn    Bedingung 2 erfüllt
        dann    wenn    Bedingung 2.1 erfüllt
                dann    Programmstück 2.1.1 ausführen
                sonst   Programmstück 2.1.2 ausführen
        sonst   wenn    Bedingung 2.2 erfüllt
                dann    Programmstück 2.2.1 ausführen
                sonst   Programmstück 2.2.2 ausführen
```

Warum die Ablaufstruktur »Kaskade« heißt, wird sehr schön aus folgendem Flußdiagramm verständlich (Bild 9.13.).

Häufiger als vollständige Verzweigungskaskaden kommen unvollständige Verzweigungskaskaden (und Mischformen mit den anderen Verzweigungsarten) vor, das sind solche, bei denen nicht alle Kombinationen von Bedingungen ausgenutzt werden. Wir sehen uns dazu gleich ein Beispiel an:

Zu lösen sei folgendes Problem: Im B-Register und C-Register stehe je eine ganze Zahl in 2-Komplement-Darstellung. Es soll eine Addition der Inhalte der beiden Register durchgeführt werden. Wenn das Ergebnis als 8-Bit-Größe darstellbar ist, soll es ins A-Register gebracht werden, sonst ins HL-Register. Das D-Register bekommt den Wert 1, wenn das Ergebnis im A-Register steht, sonst den Wert −1. Im E-Register soll das Signum des Ergebnisses der Addition untergebracht werden.

Nach Durchführung der Addition sehen wir am Zustand des Überlauf-Flags, ob ein Überlauf stattgefunden hat, das Ergebnis also nicht mit 8 Bits dargestellt werden kann. Ein Überlauf kann höchstens dann eintreten, wenn die beiden Operanden der Addition dasselbe Vorzeichen besitzen; wir überlegen uns, daß in diesem Fall genau dann auch ein Übertrag erfolgt, wenn die Operanden negativ sind. Bei eingetretenem Überlauf stehen die niederwertigen 8 Bits (LSB) des Ergebnisses bereits im A-Register und müssen nur noch ins L-Register gebracht werden.

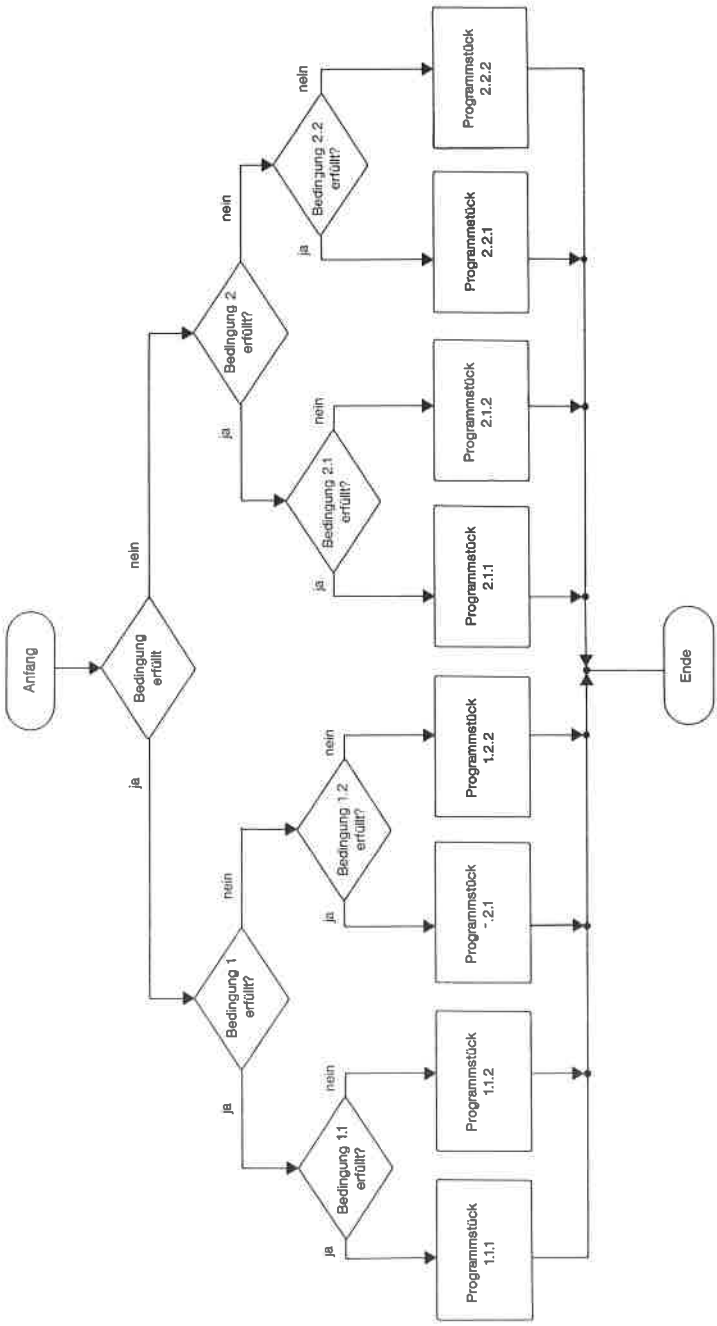


Bild 9.13. *Flußdiagramm: Schema einer Verzweigungskaskade mit drei Stufen*

Entsprechend den Regeln für Additionen im 2-Komplement wird das Übertrags-Bit in die höherwertigen 8 Bits (MSB) des Ergebnisses kopiert. Bei positivem Ergebnis (kein Übertrag) erhält das H-Register also den Wert 0, bei negativem Ergebnis den Wert FFH (oder −1).

Ob das Ergebnis eine negative 8-Bit-Größe ist, erkennen wir daran, daß das Vorzeichen-Flag gesetzt ist. Am gesetzten Null-Flag erkennen wir das Ergebnis 0. Ist weder das Überlauf-Flag noch das Vorzeichen-Flag noch das Null-Flag gesetzt, so haben wir es mit einem echt positiven 8-Bit-Ergebnis zu tun.

Wir formulieren nun eine Verzweigungskaskade entsprechend obiger Analyse und verzichten dabei zunächst einmal auf Optimierungen:

```
A <- <B> + <C>
wenn        <P> = 1
dann        wenn        <CY> = 1
            dann        H <- -1
                        L <- <A>
                        D <- -1
                        E <- -1
            sonst       H <- 0
                        L <- <A>
                        D <- -1
                        E <- 1
sonst       wenn        <S> = 1
            dann        D <- 1
                        E <- -1
            sonst       wenn    <Z> = 1
                        dann    D <- 1
                                E <- 0
                        sonst   D <- 1
                                E <- 1
```

Im Flußdiagramm dargestellt sieht der Algorithmus folgendermaßen aus (Bild 9.14.).

Das zugehörige Programm lautet:

```
ADDIER:  LD      A,B          ; 8-Bit-Arithmetik wird immer
         ADD     A,C          ; im A-Register durchgefuehrt
         JP      PO,KEINUE    ; es trat kein Ueberlauf auf
UEBERL:  JP      NC,POS16     ; positives 16-Bit-Ergebnis
NEG16:   LD      L,A          ; LSB des Ergebnisses
         LD      H,-1         ; MSB des Ergebnisses
         LD      D,-1         ; Kennung fuer 16-Bit-Ergebnis
         LD      E,-1         ; Signum einer negativen Zahl
         JP      WEITER       ; Aufgabe geloest
```

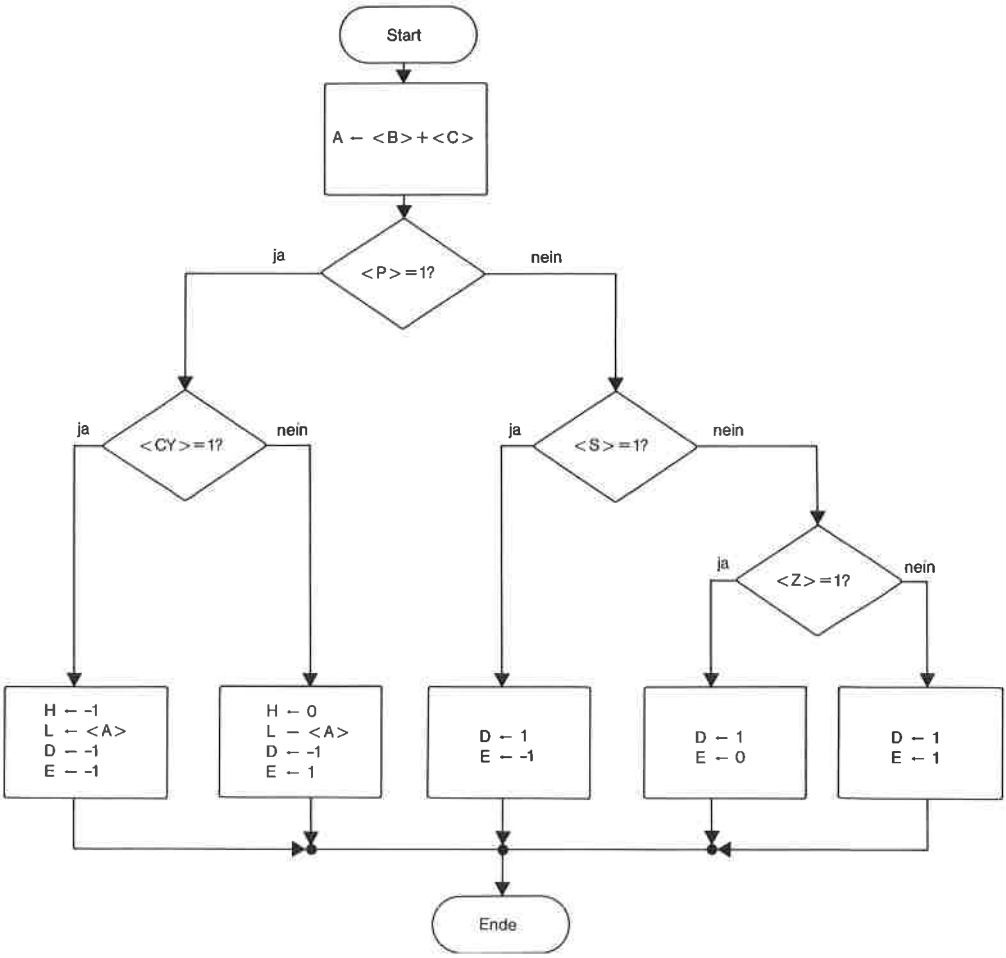


Bild 9.14. *Flußdiagramm: Beispiel einer komplizierten Addition*

```
POS16:  LD      L,A         ; LSB des Ergebnisses
        LD      H,0         ; MSB des Ergebnisses
        LD      D,-1        ; Kennung fuer 16-Bit-Ergebnis
        LD      E,1         ; Signum einer positiven Zahl
        JP      WEITER      ; Aufgabe geloest
KEINUE: JP      P,NNEG8     ; nichtnegatives 8-Bit-Ergebnis
NEG8:   LD      D,1         ; Kennung fuer 8-Bit-Ergebnis
        LD      E,-1        ; Signum einer negativen Zahl
        JP      WEITER      ; Aufgabe geloest
NNEG8:  JP      NZ,POS8     ; positives 8-Bit-Ergebnis
```

```
NULL:   LD      D,1     ; Kennung fuer 8-Bit-Ergebnis
        LD      E,0     ; Signum der Null
        JP      WEITER  ; Aufgabe geloest
POS8:   LD      D,1     ; Kennung fuer 8-Bit-Ergebnis
        LD      E,1     ; Signum einer positiven Zahl
WEITER: NOP             ; gemeinsame Fortsetzungsstelle
```

Das zugehörige Objekt-Programm ist dann:

Adresse	Objekt-Code	Marke	Anweisung	
0000	78	ADDIER:	LD	A,B
0001	81		ADD	A,C
0002	E2 1C 00		JP	PO,KEINUE
0005	D2 12 00	UEBERL:	JP	NC,POS16
0008	6F	NEG16:	LD	L,A
0009	26 FF		LD	H,-1
000B	16 FF		LD	D,-1
000D	1E FF		LD	E,-1
000F	C3 34 00		JP	WEITER
0012	6F	POS16:	LD	L,A
0013	26 00		LD	H,0
0015	16 FF		LD	D,-1
0017	1E 01		LD	E,1
0019	C3 34 00		JP	WEITER
001C	F2 26 00	KEINUE:	JP	P,NNEG8
001F	16 01	NEG8:	LD	D,1
0021	1E FF		LD	E,-1
0023	C3 34 00		JP	WEITER
0026	C2 30 00	NNEG8:	JP	NZ,POS8
0029	16 01	NULL:	LD	D,1
002B	1E 00		LD	E,0
002D	C3 34 00		JP	WEITER
0030	16 01	POS8:	LD	D,1
0032	1E 01		LD	E,1
0034	00	WEITER:	NOP	

Die Marke ADDIER dient als Ansprungpunkt für die Routine. Die Marken UEBERL, NEG16, NEG8 und NULL werden nicht benutzt; sie erleichtern aber das Verständnis des Programms und sind somit Teil der Dokumentation.

Wir wollen nun das Programm speicheroptimieren. Dazu ersetzen wir alle unbedingten absoluten Sprünge und - soweit möglich - alle bedingten absoluten Sprünge durch entsprechende relative Sprünge.

LD-Befehle mit konstantem Operanden belegen 2 Bytes, solche mit einem Register als Operanden nur 1 Byte. Wir laden deshalb die Register D und E (deren ursprünglicher Inhalt durch unser Programm ohnehin zerstört wird) mit den Werten 1 beziehungsweise −1, und speichern die Inhalte im jeweiligen Teilfall geschickt in die richtigen Register um.

Das Programmstück E <– 0 können wir, da an dieser Stelle sicher der Inhalt des A-Registers Null ist, durch E <– <A> ersetzen.

Dies zusammen liefert folgende Verzweigungskaskade:

```
D <– 1
E <– –1
A <– <B> + <C>
wenn        <P> = 1
dann        wenn        <CY> = 1
            dann        H <– <E>
                        L <– <A>
                        D <– <E>
            sonst       H <– 0
                        L <– <A>
                        D <– <E>
                        E <– 1
sonst       wenn        <S> = 1
            dann        tue nichts
            sonst       wenn    <Z> = 1
                        dann    E <– <A>
                        sonst   E <– <D>
```

Wir stellen dies auch noch im Flußdiagramm dar (Bild 9.15).

Während vorhin die Reihenfolge der LD-Befehle innerhalb jedes Teilfalls beliebig war, sind im reorganisierten Programm diese Befehle teilweise voneinander abhängig und nicht unbedingt vertauschbar.

Wir ziehen nun noch gemeinsame Befehle aus parallelen Zweigen der Kaskade nach außen:

```
D <– 1
E <– –1
A <– <B> + <C>
wenn        <P> = 1
dann        L <– <A>
            D <– <E>
            wenn        <CY> = 1
            dann        H <– <E>
            sonst       H <– 0
                        E <– 1
```

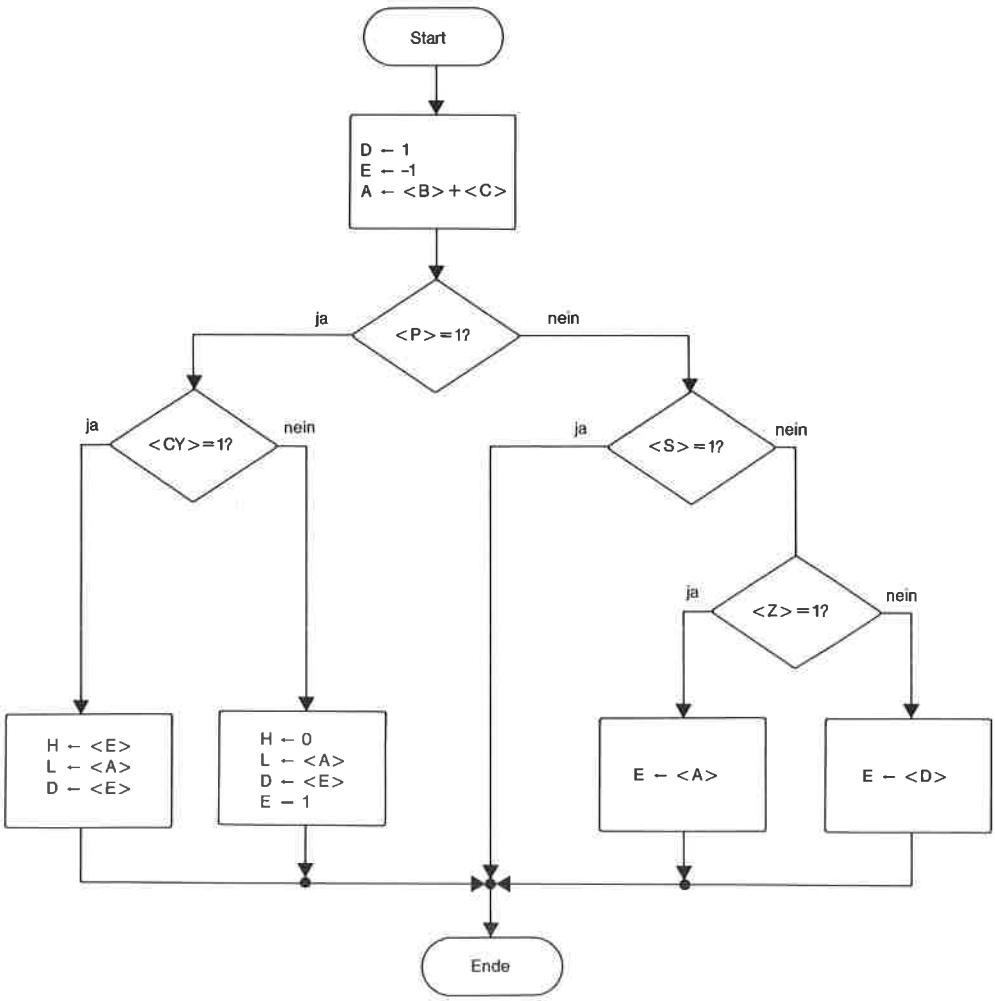


Bild 9.15. *Flußdiagramm: Komplizierte Addition (erste Optimierung)*

---

sonst	**wenn**	<S> = 1	
	dann	tue nichts	
	sonst	**wenn**	<Z> = 1
		dann	E <– <A>
		sonst	E <– <D>

Im Flußdiagramm:

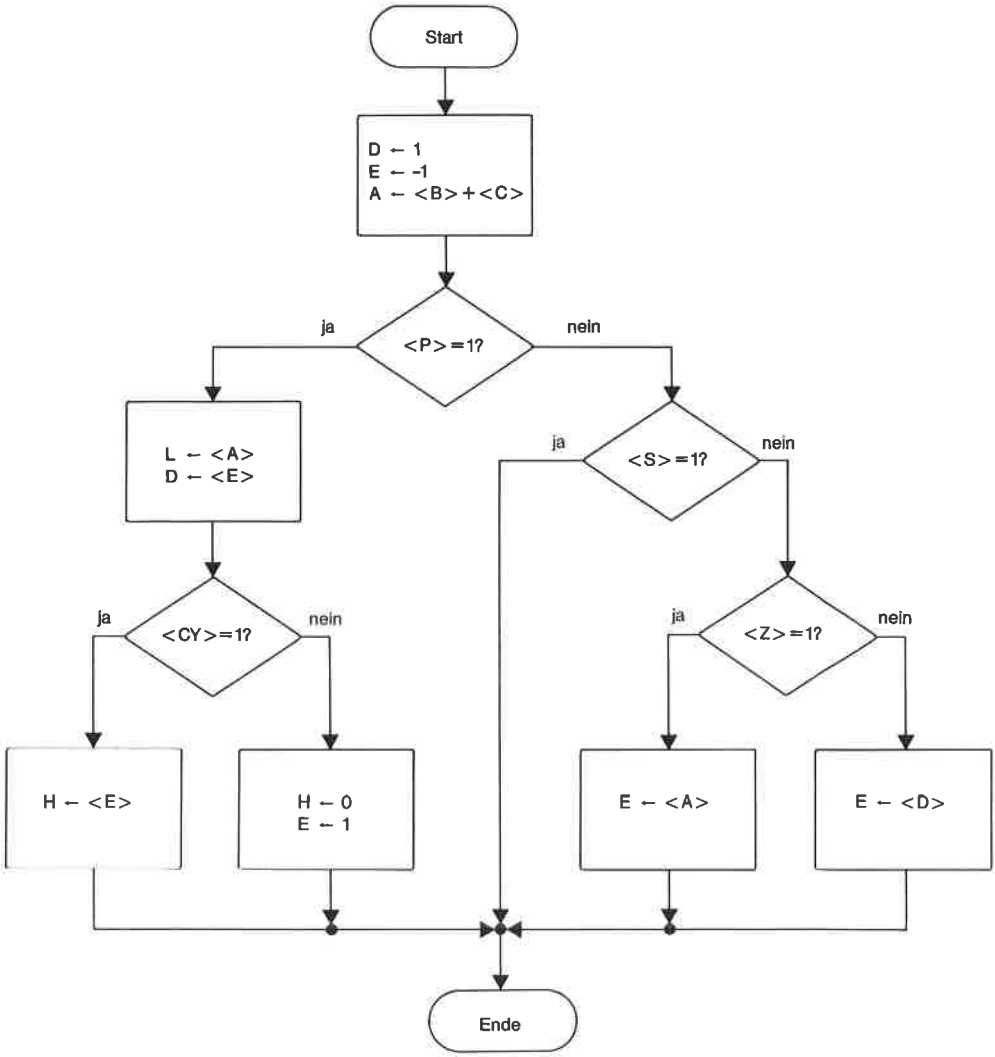


Bild 9.16. *Flußdiagramm: Komplizierte Addition (zweite Optimierung)*

Als letztes verschieben wir noch einige Befehle, um Sprünge einzusparen:

D <– 1
E <– –1
A <– <B> + <C>
wenn <P> = 1
dann L <– <A>

```
                D <– <E>
                H <– <E>
                wenn        <CY> = 1
                dann        tue nichts
                sonst       H <– 0
                            E <– 1
sonst           wenn        <S> = 1
                dann        tue nichts
                sonst       E <– <A>
                            wenn    <Z> = 1
                            dann    tue nichts
                            sonst   E <– <D>
```

Auch hierzu noch das Flußdiagramm (Bild 9.17.).

Die Optimierungen werden mit fortschreitendem Optimierungsgrad immer gefährlicher! Als Endergebnis erhalten wir folgendes Programm:

```
ADDIER:  LD      D,1          ; vorbesetzen fuer Optimierungen
         LD      E,-1         ; vorbesetzen fuer Optimierungen
         LD      A,B          ; 8-Bit-Arithmetik wird immer
         ADD     A,C          ; im A-Register durchgefuehrt
         JP      PO,KEINUE    ; es trat kein Ueberlauf auf
UEBERL:  LD      L,A          ; LSB des Ergebnisses
         LD      D,E          ; Kennung fuer 16-Bit-Ergebnis
         LD      H,E          ; MSB des Ergebnisses mit -1
                              ; vorbesetzen fuer Optimierung
         JR      C,WEITER     ; negatives 16-Bit-Ergebnis
                              ; Aufgabe geloest
POS16:   LD      H,0          ; MSB des Ergebnisses
         LD      E,1          ; Signum einer positiven Zahl
         JR      WEITER       ; Aufgabe geloest
KEINUE:  JP      M,WEITER     ; negatives 8-Bit-Ergebnis
                              ; Aufgabe geloest
NNEG8:   LD      E,A          ; Signum des Ergebnisses
                              ; gegebenenfalls mit 0
                              ; vorbesetzen fuer Optimierung
         JR      Z,WEITER     ; Ergebnis Null, Aufgabe geloest
POS8:    LD      E,D          ; Signum einer positiven Zahl
WEITER:  NOP                  ; gemeinsame Fortsetzungsstelle
```

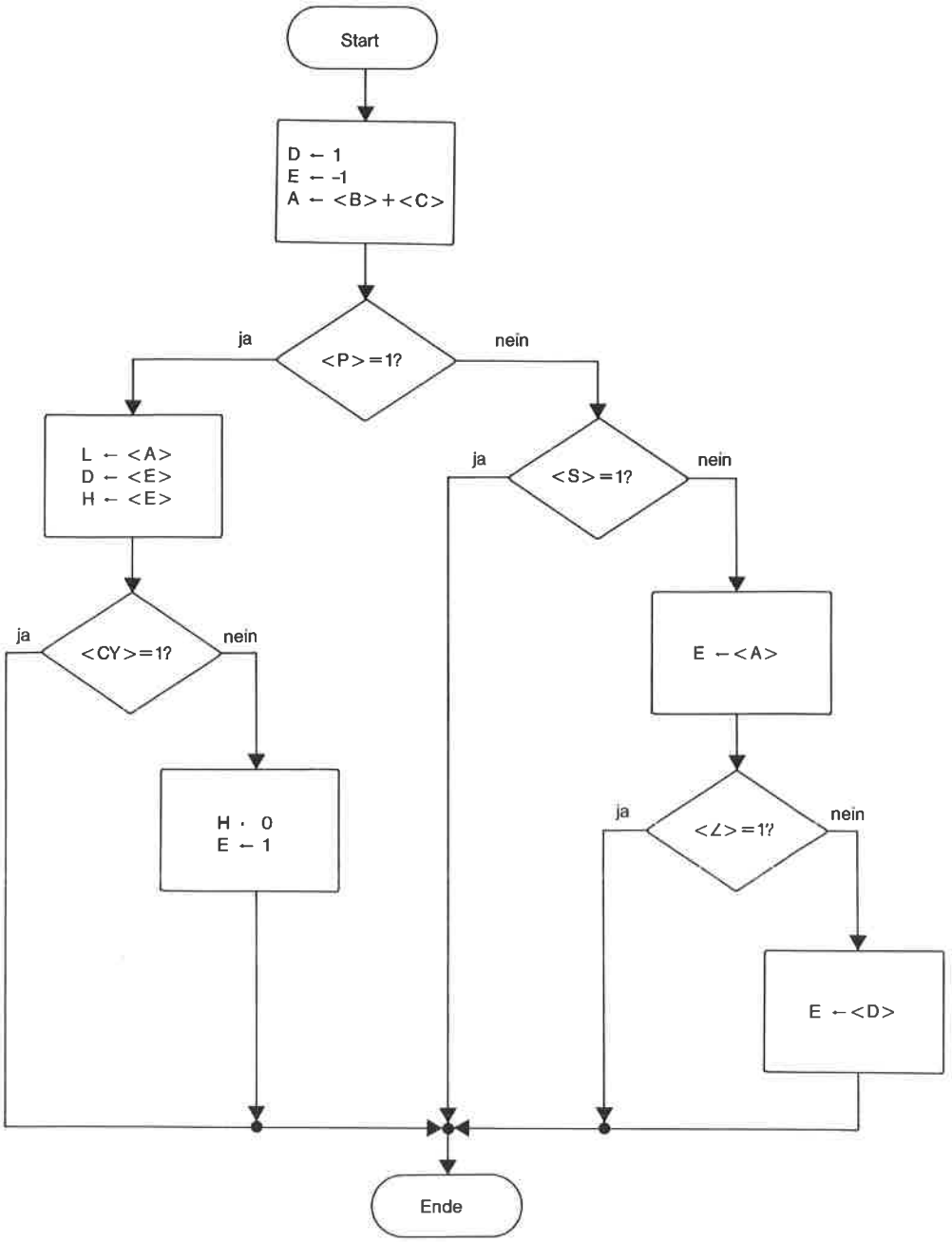


Bild 9.17. *Flußdiagramm: Komplizierte Addition (dritte Optimierung)*

Der zugehörige Objekt-Code lautet:

Adresse	Objekt-Code	Marke	Anweisung	
0000	16 01	ADDIER:	LD	D,1
0002	1E FF		LD	E,−1
0004	78		LD	A,B
0005	81		ADD	A,C
0006	E2 14 00		JP	PO,KEINUE
0009	6F	UEBERL:	LD	L,A
000A	53		LD	D,E
000B	63		LD	H,E
000C	38 0D		JR	C,WEITER
000E	26 00	POS16:	LD	H,0
0010	1E 01		LD	E,1
0012	18 07		JR	WEITER
0014	FA 1B 00	KEINUE:	JP	M,WEITER
0017	5F	NNEG8:	LD	E,A
0018	28 01		JR	Z,WEITER
001A	5A	POS8:	LD	E,D
001B	00	WEITER:	NOP	

Von den ursprünglichen 52 Bytes (der NOP-Befehl wird nicht mitgezählt) haben wir somit 25 Bytes wegoptimiert, das sind immerhin fast 50%! (Wer für das Problem mit weniger als 27 Bytes auskommt, darf sich schon jetzt **Z80-Programmierer** nennen!)

Übungen

1. Schreibe auch für die Algorithmen der ersten und zweiten Optimierungsstufe die zugehörigen Programme. Teste alle Varianten sorgfältig aus.

2. Wenn Sie alle drei Optimierungen genau verstanden haben, können Sie versuchen, das Programm noch kürzer zu gestalten!

3. Wir wollen ein Teilstück einer Cursor-Steuerung (zum Beispiel für ein Spielprogramm) realisieren.
   Die Zeilen des Bildschirms seien von 0 bis 23 numeriert, die Spalten von 0 bis 79. Die Position des Cursors (dies ist ein meist blinkendes spezielles Zeichen, das dem Benutzer die aktuelle Eingabeposition am Bildschirm anzeigt) soll im BC-Register gehalten werden (Zeilennummer im B-Register, Spaltennummer im C-Register).
   Wir nehmen an, daß vom numerischen Tastenfeld des Computers eines der Zeichen '1' bis

'9' ins A-Register eingelesen wurde. Dieses Zeichen soll nun als Befehl an die Cursor-Steuerung interpretiert werden. Dabei bedeutet

'1'	ein Zeichen nach links unten
'2'	ein Zeichen nach unten
'3'	ein Zeichen nach rechts unten
'4'	ein Zeichen nach links
'5'	auf Position (0,0) springen
'6'	ein Zeichen nach rechts
'7'	ein Zeichen nach links oben
'8'	ein Zeichen nach oben
'9'	ein Zeichen nach rechts oben

Die Bedeutung dieser Codierung resultiert daraus, daß auf vielen Computern mit numerischem Tastenfeld die Tasten folgendermaßen angeordnet sind:

7	8	9
4	5	6
1	2	3
	0	

Der Cursor soll am Rand des Bildschirms anhalten, also seinen zugewiesenen Bereich nicht über den Rand verlassen.
Schreibe nun ein Programm, welches das BC-Register dem Wert des A-Registers entsprechend neu setzt.

4. Modifiziere das Programm aus Aufgabe 3 so, daß der Cursor den Bildschirm über den Rand verlassen und an der gegenüberliegenden Position wieder betreten kann (engl. wrap around).

10
Worte

Ein Wort (engl. word) ist eine Datenstruktur, bestehend aus zwei Bytes. Worten wird – wie wir es schon von den Bytes her kennen – meist eine Interpretation unterschoben. Besonders häufig ist die Interpretation als binär-codierte vorzeichenlose ganze Zahl (Zahlbereich 0 bis 65535) oder als ganze Zahl in 2-Komplement-Darstellung (Zahlbereich - 32768 bis +32767).

Dasjenige Byte eines Worts, das den höherwertigen Anteil der dargestellten Zahl enthält, heißt MSB (most significant byte), das andere Byte heißt LSB (least significant byte). Im Speicher werden MSB und LSB normalerweise fortlaufend abgelegt, wobei das MSB üblicherweise die höhere Adresse erhält. Wir wollen uns an diese Konvention halten, da der Z80 Wort-Operationen stets in dieser Weise ausführt.

Ein Wort kann auch in einem der Doppelregister, in einem Indexregister oder im Stapelzeiger untergebracht werden (auch im Befehlszähler kann ein Wort untergebracht werden; da der Inhalt des Befehlszählers die Adresse des nächsten auszuführenden Befehls enthält, stellt das Laden des Befehlszählers mit einem neuen Inhalt einen Sprung innerhalb des Programms dar). Wir werden uns vorläufig nur mit Worten beschäftigen, die im Speicher oder in einem der Registerpaare BC, DE, HL untergebracht sind.

10.1 Ladebefehle für Worte

Wenn wir eines der genannten Registerpaare mit einem bestimmten 16-Bit-Wert laden wollen, so können wir dies im Prinzip mit den bereits bekannten LD-Befehlen für 8-Bit-Register tun. Zum Beispiel soll der Wert 21F7H ins DE-Register gebracht werden:

```
LD      D,21H
LD      E,0F7H      ; fuehrende 0 nicht vergessen!
```

Der Objekt-Code dazu lautet dann:

Adresse	Objekt-Code	Marke	Anweisung	
0000	16 21		LD	D,21H
0002	1E F7		LD	E,0F7H

Schneller (und sogar mit kürzerem Objekt-Code) erreichen wir dies durch einen 16-Bit-LD-Befehl:

```
        LD          DE,21F7H    ; Konstante 21F7H vom Typ Wort
                                ; in Doppelregister DE laden
```

Der Objekt-Code dieses Befehls lautet nämlich:

Adresse	Objekt-Code	Marke	Anweisung	
0000	11 F7 21		LD	DE,21F7H

Achte genau auf die Anordnung der beiden Bytes im Objekt-Code!

Leider gibt es keine Befehle, um ein Doppelregister direkt in ein anderes Doppelregister umzuspeichern. Wenn wir jetzt zum Beispiel das DE-Register ins HL-Register übertragen wollen, so müssen wir schreiben:

```
        LD          H,D         ; Inhalt des DE-Registers
        LD          L,E         ; ins HL-Register bringen
```

Merke: Bei Worten in Doppelregistern enthält das zuerst genannte Register das MSB, das anschließend genannte Register das LSB!

Das Laden eines Doppelregisters mit einem Wort, das im Speicher steht, realisieren wir dagegen wieder durch einen einzigen Befehl. Wir müssen dazu die Speicheradresse des LSB spezifizieren. Beispiel: Unter der Adresse 31A7H steht der 8-Bit-Wert E9H, unter der Adresse 31A8H der 8-Bit-Wert 23H. Der Befehl

```
        LD          BC,(31A7H) ; Lade Inhalt der Adresse 31A7H
                               ; ins C-Register und Inhalt der
                               ; Adresse 31A8H ins B-Register
```

bewirkt dann, daß das B-Register den Wert 23H erhält, das C-Register den Wert E9H. Als Registerpaar gesehen enthält das BC-Register damit den Wert 23E9H.

Der Objekt-Code des Befehls lautet:

Adresse	Objekt-Code	Marke	Anweisung	
0000	ED 4B A7 31		LD	BC,(31A7H)

Interessanterweise belegt der Objekt-Code des Befehls

```
        LD      HL,(adresse)
```

nur 3 Bytes, während die Befehle

```
        LD      BC,(adresse)
```

und

```
        LD      DE,(adresse)
```

je 4 Bytes Speicherplatz benötigen. Dies liegt daran, daß letztere beim direkten Vorgänger des Z80 - dem Prozessor 8080 von INTEL - noch nicht vorhanden waren, im Gegensatz zum erstgenannten Befehl.

Wenn wir umgekehrt den Inhalt eines Doppelregisters in den Speicher kopieren wollen, so benutzen wir dazu einen Befehl der Form

```
        LD      (adresse),registerpaar
```

Die Pseudo-Operation EQU, dic wir im Kapitel »Verzweigungsketten« kennengelernt haben, können wir auch benutzen, um 16-Bit-Konstanten zu vereinbaren; dies sind je nach Anwendung Worte oder Adressen. Zum Beispiel:

```
UMSATZ  EQU     1E94H       ; Adresse
STEUER  EQU     1E9BH       ; Adresse
KOSTEN  EQU     23400       ; Konstante vom Typ Wort

        LD      (STEUER),BC ; Wort in den Speicher bringen
        LD      HL,(UMSATZ) ; Wort aus dem Speicher holen
        LD      DE,KOSTEN   ; Konstante in Register schreiben
```

Übungen

1. Lade die Zahl 22131 ins BC-Register.

2. Speichere das BC-Register ins DE-Register um.

3. Lade das ab der Adresse 2124H stehende Wort ins DE-Register.

4. Speichere das ab Adresse 4256H stehende Wort auch unter der Adresse 567BH ab.

10.2 Vereinbarung von Variablen durch Pseudo-Operationen

Bisher haben wir bei Operationen auf dem Speicher mehr oder weniger willkürlich Speicheradressen gewählt. Richtige Variablen gehören aber zu einem bestimmten Objektprogramm und werden in diesem mit Hilfe der Pseudo-Operationen **DEFB** (define byte) und **DEFW** (define word) vereinbart. Zum Beispiel bewirkt die (mit einer Marke versehene) Pseudo-Operation

```
ZEICH:    DEFB      2AH       ; 1 Byte Speicherplatz unter
                                ; dem Namen ZEICH reservieren
                                ; und mit dem Wert 2AH belegen
```

daß ein Speicherplatz der Größe 1 Byte reserviert wird, der den Initialwert 2AH erhält und unter der symbolischen Adresse (Variablenname!) ZEICH angesprochen werden kann. Ebensogut hätten wir

```
ZEICH:    DEFB      '*'       ; 1 Byte Speicherplatz unter
                                ; dem Namen ZEICH reservieren
                                ; und mit dem Wert '*' belegen
```

dafür schreiben können, denn der Objekt-Code lautet in beiden Fällen (unter der Annahme, daß ZEICH die Adresse 72A4H bedeutet):

Adresse	Objekt-Code	Marke	Anweisung
72A4	2A	ZEICH:	DEFB '*'

Zur Reservierung eines Speicherplatzes für ein Wort mit Initialwert 1719H und symbolischer Adresse DAUER schreiben wir die Pseudo-Operation

```
DAUER:    DEFW      1719H     ; 1 Wort Speicherplatz unter
                                ; dem Namen DAUER reservieren
                                ; und mit dem Wert 1719H belegen
```

An der Adresse DAUER wird der Wert 19H abgelegt, an der Adresse DAUER+1 wird der Wert 17H abgelegt. Wenn DAUER die Adresse 274BH symbolisiert, lautet der Objekt-Code:

Adresse	Objekt-Code	Marke	Anweisung
274B	19 17	DAUER:	DEFW 1719H

Was wir mit den Pseudo-Operationen DEFB und DEFW bisher realisiert haben, waren »initialisierte Variablen«, also Variablen, denen vor dem Start des Programms bereits Werte zugewiesen wurden. Nun kommt es aber auch vor, daß wir Variablen im Programm benötigen, die zunächst keinen Wert haben sollen, sondern diesen erst im Verlauf des Programms zugewiesen bekommen. Diese uninitialisierten Variablen schaffen wir uns mit der Pseudo-Operation **DEFS** (define storage). Als Operand der DEFS-Pseudo-Operation wird die Anzahl der Bytes angegeben, die für die Variable zur Verfügung gestellt werden sollen. Betrachte dazu folgende Beispiele:

```
AlPHA:    DEFS    1       ; Variable vom Type Byte
BETA:     DEFS    2       ; Variable vom Type Wort
GAMMA:    DEFS    8       ; Variable mit einem
                          ; Speicherbedarf von 8 Bytes,
                          ; zum Beispiel fuer
                          ; eine Gleitpunktzahl
DELTA:    DEFS    256     ; Variable mit einem
                          ; Speicherbedarf von 256 Bytes,
                          ; zum Beispiel fuer
                          ; eine Zeichenreihe, einen
                          ; Puffer oder eine Matrix
```

Relativ häufig müssen wir Texte in Variablen abspeichern. Zur Vereinbarung solcher initialisierter Text-Variablen gibt es die Pseudo-Operation **DEFM** (define memory), mit der eine Variable angelegt wird, deren Größe der angegebenen Zeichenkette angepaßt ist:

```
SPRUCH:   DEFM    'Dies ist eine schoene Zeichenkette!'
```

Die Variable SPRUCH belegt nun 35 Bytes im Speicher.

Die Adressen für den Objekt-Code und für Datenspeicherplätze werden der Auflistung im Quellprogramm entsprechend vom Assembler fortlaufend vergeben. Befehle und Pseudo-Operationen können dabei in beliebiger Reihenfolge auftreten. Es ist jedoch guter Programmierstil, Daten und Code je in einem zusammenhängenden Bereich zu vereinbaren!

Übungen

1. Vereinbare initialisierte Variablen für das Byte 45H, das Zeichen '+', das Wort 1700 und die Zeichenreihe »Happy New Year!«. Wieviel Bytes Speicherplatz belegen die einzelnen Variablen?

2. Vereinbare uninitialisierte Variablen für ein Byte, ein Wort, eine 32-Bit-Zahl, eine Zeichenkette mit 7 Zeichen, einen Puffer mit einer Länge von 128 Bytes.

10.3 Arithmetik mit Worten

Arithmetische Operationen mit Worten werden stets im HL-Register ausgeführt, so wie für Bytes im A-Register. Wir sehen uns gleich einige Beispiele dazu an:

Wir wollen eine Routine schreiben, die eine vorzeichenlose 8-Bit-Größe mit der Konstanten 10 multipliziert. Dabei kann ein Ergebnis entstehen, das nicht als 8-Bit-Größe darstellbar ist; für die Aufnahme des Ergebnisses stellen wir deshalb einen Speicherplatz zur Ablage eines Datenwerts vom Typ »Wort« zur Verfügung. Für den Operanden richten wir einen Speicherplatz zur Aufnahme eines Datenwerts vom Typ »Byte« ein. Die Multiplikation führen wir - wie im Kapitel Sequenzen für 8-Bit-Größen - durch mehrmalige Addition aus.

Vor Durchführung der Multiplikation expandieren wir den Operanden vom Typ »Byte« zum Typ »Wort«, und speichern ihn gleich in den Akkumulator, das HL-Register. Eine Kopie des Operanden halten wir zusätzlich noch im DE-Register. Also:

```
OPERAN:   DEFS      1             ; Speicherplatz fuer Operand
                                  ; Wert wird spaeter eingesetzt
ERGEBN:   DEFS      2             ; Speicherplatz fuer Ergebnis
MULT10:   LD        A,(OPERAN)    ; Operand beschaffen
          LD        L,A
          LD        H,0           ; Operand zu Wort expandieren
          LD        D,H           ; Kopie des Operanden
          LD        E,L           ; erstellen
          ADD       HL,HL         ; Operand verdoppeln
          ADD       HL,HL         ; Operand vervierfachen
          ADD       HL,DE         ; Operand verfuenffachen
          ADD       HL,HL         ; Operand verzehnfachen
          LD        (ERGEBN),HL   ; Ergebnis abspeichern
```

Die weitere Arithmetik auf Größen vom Typ »Wort« ist recht dürftig ausgelegt: Es gibt einen Befehl **ADC** (add carry), der wie der ADD-Befehl wirkt, jedoch auch noch den Inhalt des Übertrag-Flags zum Ergebnis addiert (diesen Befehl werden wir später beim Aufbau einer komplizierten Arithmetik verwenden). Schließlich existiert noch der Befehl **SBC** (subtract carry), der den Inhalt eines Doppelregisters vom HL-Register subtrahiert und anschließend auch noch den Wert des Übertrag-Flags abzieht. Um mit dem SBC-Befehl normale Subtraktionen durchzuführen, bedienen wir uns des Befehls **SCF** (set carry flag), der den Wert 1 ins Übertrag-Flag bringt, und des Befehls **CCF** (complement carry flag), der den Wert des Übertrag-Flags durch sein 1-Komplement ersetzt (in Kapitel 12 lernen wir einen Trick kennen, das Löschen des Übertrag-Flags kürzer zu bewerkstelligen). Wir sehen uns den Ablauf einer Subtraktion von Worten exemplarisch an:

```
OP1:      DEFS      2          ; Speicherplatz fuer
                               ; ersten Operanden
OP2:      DEFS      2          ; Speicherplatz fuer
                               ; zweiten Operanden
ERGEBN:   DEFS      2          ; Speicherplatz fuer Ergebnis
SUBTRA:   LD        HL,(OP1)   ; 1. Operand holen
          LD        DE,(OP2)   ; 2. Operand holen
          SCF                  ; Uebertrag-Flag setzen
          CCF                  ; Uebertrag-Flag loeschen
          SBC       HL,DE      ; normale Subtraktion ausfuehren
          LD        (ERGEBN),HL ; Ergebnis abspeichern
```

Der Umgang mit dem ADC-Befehl ist ganz analog dazu. Wir sehen uns folgendes Beispiel an: Zwei vorzeichenlose ganze Zahlen, die als 32-Bit-Größen im Speicher dargestellt sind, sollen addiert und das Ergebnis (modulo 232) ebenfalls im Speicher untergebracht werden. Wir führen sukzessive zwei Additionen mit Worten aus. Zuerst addieren wir die niederwertigen 16 Bits der beiden Zahlen und speichern das Ergebnis als die niederwertigen 16 Bits des Resultats ab. Dann addieren wir - unter Einbeziehung eines eventuell angefallenen Übertrags - die höherwertigen 16 Bits der Zahlen und speichern das Ergebnis als die höherwertigen 16 Bits des Resultats ab. Den zuletzt angefallenen Übertrag brauchen wir nicht zu berücksichtigen, da wir modulo 232 reduzieren wollen. Das zugehörige Programm lautet nun:

```
OP1:      DEFS      4          ; Speicherplatz fuer
                               ; ersten Operanden
OP2:      DEFS      4          ; Speicherplatz fuer
                               ; zweiten Operanden
ERGEBN:   DEFS      4          ; Speicherplatz fuer Ergebnis
ADD32:    LD        HL,(OP1)   ; niederwertige 16 Bits des
                               ; ersten Operanden holen
          LD        DE,(OP2)   ; niederwertige 16 Bits des
                               ; zweiten Operanden holen
          ADD       HL,DE      ; niederwertige 16 Bits des
                               ; Ergebnisses berechnen
          LD        (ERGEBN),HL ; niederwertige 16 Bits des
                               ; Ergebnisses abspeichern
          LD        HL,(OP1+2) ; hoeherwertige 16 Bits des
                               ; ersten Operanden holen
          LD        DE,(OP2+2) ; hoeherwertige 16 Bits des
                               ; zweiten Operanden holen
          ADC       HL,DE      ; hoeherwertige 16 Bits des
                               ; Ergebnisses berechnen, dabei
                               ; Uebertrag aus vorhergehender
                               ; Addition beruecksichtigen
```

```
        LD          (ERGEBN+2),HL   ; hoeherwertige 16 Bits des
                                    ; Ergebnisses abspeichern
```

Sehen Sie sich genau an, wie die niederwertigen beziehungsweise höherwertigen 16 Bits der jeweiligen Zahlen adressiert werden!

Während die Befehle ADC und SBC das Überlauf-Flag beeinflussen und somit auch für 2-Komplements-Arithmetik einsetzbar sind, werden die Befehle ADD, INC und DEC (letztere lernen wir im nächsten Kapitel kennen) auf Worten hauptsächlich für Adreßberechnungen verwendet.

Merke: Arithmetische Operationen auf Worten finden stets im HL-Register statt!

Übungen

1. Schreibe ein Programm, welches das 12fache einer mit 8 Bits im Speicher dargestellten vorzeichenlosen ganzen Zahl berechnet und das Ergebnis mit 16 Bits wieder im Speicher ablegt.

2. Schreibe ein Programm, das zwei im Speicher stehende Binärzahlen mit je 64 Bits addiert und das Ergebnis - modulo 264 reduziert - im Speicher ablegt.

3. Schreibe ein Programm, das eine in 2-Komplement-Darstellung mit 64 Bits im Speicher stehende ganze Zahl von einer anderen solchen Zahl subtrahiert und das Ergebnis (ohne Berücksichtigung eines eventuell aufgetretenen Überlaufs) wieder mit 64 Bits im Speicher darstellt.

11
Adressen und Zeiger

Unter einer Adresse verstehen wir eine Speicheradresse, beim Z80 also eine Größe vom Typ »Wort«. Adressen lassen sich unterscheiden in Code-Adressen und Daten-Adressen. Code-Adressen sind Anfangsadressen von Befehlen; sie werden benutzt, um einen bestimmten Befehl anzuspringen. Daten-Adressen sind Adressen von Datenwerten im Speicher.

Ein Zeiger ist ein von den eigentlichen Daten unabhängiger Verweis auf einen Datenwert. Auf dem Z80 repräsentiert man einen Zeiger am einfachsten durch die Anfangsadresse der (möglicherweise kompliziert strukturierten) Datenwerte. Zeiger dienen häufig zur Verkettung der Elemente dynamischer Datenstrukturen.

11.1 Indirekte Sprünge

Code-Adressen sind uns bereits in Form direkter Sprungadressen im JP-Befehl begegnet (zum Beispiel in JP 3A4BH oder in JP WEITER). Enthält das HL-Register eine Code-Adresse (man nennt HL dann ein Code-Adreß-Register), so können wir auch einen indirekten Sprung auf diese Adresse ausführen mittels

```
JP          (HL)        ; lade PC-Register mit dem
                        ; Inhalt des HL-Registers,
                        ; d.h. springe auf den Befehl,
                        ; dessen Anfangsadresse
                        ; im HL-Register steht
```

Wir betrachten dazu folgendes Beispiel: Es soll ein Programmstück angesprungen werden, dessen Anfangsadresse ab der Adresse 5484H im Speicher steht. Also:

```
        LD      HL,(5484H) ; Code-Adresse holen
        JP      (HL)       ; Programmstück anspringen
```

Als Objekt-Code ergibt sich:

Adresse	Objekt-Code	Marke	Anweisung	
0000	2A 84 54		LD	HL,(5484H)
0003	E9		JP	(HL)

Die Indexregister IX und IY können ebenfalls als Code-Adreß-Register benutzt werden:

```
        JP      (IX)       ; lade PC-Register mit dem
                           ; Inhalt des IX-Registers,
                           ; d.h. springe auf den Befehl,
                           ; dessen Anfangsadresse
                           ; im IX-Register steht
```

beziehungsweise

```
        JP      (IY)       ; lade PC-Register mit dem
                           ; Inhalt des IY-Registers,
                           ; d.h. springe auf den Befehl,
                           ; dessen Anfangsadresse
                           ; im IY-Register steht
```

Da wir die Indexregister in den Kapiteln »Verbunde« und »Verschiebbare Programme« intensiv studieren werden, sei für den Umgang mit Indexregistern zunächst auf diese Kapitel verwiesen.

Übungen

1. Es soll ein Programmstück angesprungen werden, dessen Anfangsadresse relativ zum Inhalt des HL-Registers durch den Inhalt des BC-Registers gegeben ist (eine Form des relativen indirekten Sprungs).

11.2 Indirekte Adressierung von Daten

Auch Daten-Adressen haben wir in Form direkter Adressierung schon kennengelernt (zum Beispiel in LD A, (1516H) oder in LD (ERGEBN), HL). Zur indirekten Adressierung von Datenwerten werden die Register BC, DE, HL, SP, IX und IY benutzt (bezüglich IX und IY als Daten-Adreß-Register sei auf das Kapitel »Verbunde« verwiesen; auf den Stapel-Zeiger SP werden wir im Kapitel »Stapel« näher eingehen).

Die Register BC und DE haben nur sehr beschränkte Einsatzmöglichkeiten als Daten-Adreß-Register. Es gibt je zwei Lade-Befehle, nämlich:

```
LD      A,(BC)      ; Inhalt der Speicherzelle,
                    ; deren Adresse im BC-Register
                    ; steht, ins A-Register bringen
LD      A,(DE)      ; Inhalt der Speicherzelle,
                    ; deren Adresse im DE-Register
                    ; steht, ins A-Register bringen
LD      (BC),A      ; Inhalt des A-Registers in die
                    ; Speicherzelle bringen, deren
                    ; Adresse im BC-Register steht
LD      (DE),A      ; Inhalt des A-Registers in die
                    ; Speicherzelle bringen, deren
                    ; Adresse im DE-Register steht
```

Wir bringen gleich ein Anwendungsbeispiel: Eine im Speicher stehende ganze Zahl in 2-Komplement-Darstellung soll negiert und das Ergebnis in einer anderen Speicherzelle untergebracht werden. Die Adresse des Operanden erwarten wir dabei im BC-Register, die Adresse des Ergebnisses im DE-Register. Das Programm lautet damit:

```
LD      A,(BC)      ; Operand aus Speicher holen
NEG                 ; Operation durchfuehren
LD      (DE),A      ; Ergebnis abspeichern
```

Darüber hinaus gibt es noch spezielle Befehle für blockweises Bewegen von Daten, in denen das DE-Register als Daten-Adreß-Register eingesetzt wird (diese Befehle werden wir im Kapitel »Felder« kennenlernen).

Wichtigstes Register für die Daten-Adressierung ist das HL-Register. In vielen Befehlen kann statt eines 8-Bit-Registers auch eine im Speicher stehende, durch das HL-Register indirekt adressierte 8-Bit-Größe auftauchen, zum Beispiel in

```
LD      C,(HL)      ; Inhalt der Speicherzelle,
                    ; deren Adresse im HL-Register
                    ; steht, ins C-Register bringen
LD      (HL),E      ; Inhalt des E-Registers in die
                    ; Speicherzelle bringen, deren
                    ; Adresse im HL-Register steht
LD      (HL),24H    ; Den Wert 24H in die
                    ; Speicherzelle bringen, deren
                    ; Adresse im HL-Register steht
ADD     A,(HL)      ; Inhalt der Speicherzelle,
                    ; deren Adresse im HL-Register
                    ; steht, zum A-Register addieren
```

```
SUB     (HL)    ; Inhalt der Speicherzelle,
                ; deren Adresse im HL-Register
                ; steht, von A-Register abziehen
CP      (HL)    ; Inhalt der Speicherzelle,
                ; deren Adresse im HL-Register
                ; steht, mit dem Inhalt des
                ; A-Registers vergleichen
```

Die gesamte Bandbreite von Anwendungsmöglichkeiten des HL-Registers als Daten-Adreß-Register kannst Du in Anhang A nachlesen!

Wir betrachten nun ein Problem, bei dem wir drei Daten-Adreß-Register gleichzeitig benötigen: Zwei im Speicher stehende vorzeichenlose ganze Zahlen mit je 8 Bits sollen addiert und das Ergebnis – reduziert modulo 256 – im Speicher deponiert werden. Die drei Speicheradressen sollen jeweils über ein Doppelregister spezifiziert sein. Bei solchen 3-Adreß-Operationen muß normalerweise der zweite Operand durch das HL-Register angegeben werden, damit er direkt (also ohne Zwischenspeicherung in einem Register) in die Operation einbezogen werden kann; die Zuordnung von BC-Register und DE-Register zu erstem Operanden und Ergebnis ist dagegen beliebig (wir wählen das BC-Register als Adreß-Register für das Ergebnis, das DE-Register als Adreß-Register für den ersten Operanden):

```
LD      A,(DE)  ; ersten Operanden holen
ADD     A,(HL)  ; zweiten Operanden direkt in
                ; die Operation einbeziehen
LD      (BC),A  ; Ergebnis abspeichern
```

Der erzeugte Objekt-Code lautet (er ist mit 3 Bytes extrem effizient!):

Adresse	Objekt-Code	Marke	Anweisung	
0000	1A		LD	A,(DE)
0001	86		ADD	A,(HL)
0002	02		LD	(BC),A

Im vorhergehenden Kapitel haben wir gesehen, wie man mit Größen vom Typ »Wort« rechnen kann. Diese Methoden können wir damit zur Berechnung von Daten-Adressen benutzen. Meist stehen jedoch die Daten fortlaufend im Speicher, sei es, weil mehrere Datenwerte logisch zusammengehören, oder weil ein Datenwert mehr als ein Byte Speicherplatz benötigt. In diesem Fall liegen die verwendeten Adressen nahe beieinander; der Z80 unterstützt diesen Sachverhalt durch die Befehle **INC** (increment) und **DEC** (decrement), deren Verwendung weniger umständlich als die Berechnung mit Hilfe von Wort-Arithmetik ist. Der INC-Befehl vergrößert den Inhalt eines Doppelregisters um 1, der DEC-Befehl verkleinert ihn um 1, also zum Beispiel

```
        INC     HL      ; Inhalt des HL-Registers
                        ; um 1 vergroeßern
        DEC     BC      ; Inhalt des BC-Registers
                        ; um 1 verkleinern
```

Die Befehle sind schneller, kürzer und klarer als Arithmetik-Befehle; es wird auch kein zweites Register zur Berechnung benötigt. Daß INC und DEC keine echten Arithmetik-Befehle sind, erkennt man daran, daß die Flags nicht verändert werden (die Herstellerfirma rechnet sie allerdings doch zu den Arithmetik-Befehlen, was sich in der Klassifikation im Anhang niederschlägt).

Wir demonstrieren nun die Überlegenheit der indirekten Adressierung gegenüber der direkten Adressierung an einem Beispiel: Wir wollen eine vorzeichenbehaftete ganze Zahl mit 8 Bits in 2-Komplement-Darstellung von einer anderen solchen Zahl subtrahieren. Die beiden Datenwerte sollen an festen Adressen im Speicher stehen, das Ergebnis (ohne Berücksichtigung von Überlauf) soll ebenfalls in den Speicher gebracht werden. Unser Datenbereich sehe also folgendermaßen aus:

```
OP1:    DEFS    1       ; Speicherplatz fuer
                        ; ersten Operanden
OP2:    DEFS    1       ; Speicherplatz fuer
                        ; zweiten Operanden
ERG:    DEFS    1       ; Speicherplatz fuer Ergebnis
```

Wir zeigen zuerst eine Realisierung mit direkter Adressierung:

```
        LD      A,(OP2) ; zweiter Operand muss zuerst
                        ; geholt werden, da als Ziel-
                        ; register nur das A-Register
                        ; zur Verfuegung steht
        LD      D,A     ; zweiten Operanden hilfsweise
                        ; in Register sichern
        LD      A,(OP1) ; ersten Operanden holen
        SUB     D       ; Operation durchfuehren
        LD      (ERG),A ; Ergebnis abspeichern
```

Das zugehörige Objekt-Programm lautet dann:

Adresse	Objekt-Code	Marke	Anweisung	
0000		OP1:	DEFS	1
0001		OP2:	DEFS	1
0002		ERG:	DEFS	1
0003	3A 01 00		LD	A,(OP2)

```
0006    57          LD      D,A
0007    3A 00 00    LD      A,(OP1)
000A    92          SUB     D
000B    32 02 00    LD      (ERG),A
```

Das Programmstück belegt 11 Bytes und ist abhängig von den fest vorgegebenen Speicher-Adressen.

Alternativ dazu nun eine Lösung mittels indirekter Adressierung:

```
        LD      HL,OP1      ; Anfangsadresse des
                            ; Datenbereichs in das
                            ; Daten-Adress-Register laden
        LD      A,(HL)      ; ersten Operanden holen
        INC     HL          ; auf zweiten Operanden zeigen
        SUB     (HL)        ; zweiten Operanden subtrahieren
        INC     HL          ; auf Speicherplatz
                            ; fuer Ergebnis zeigen
        LD      (HL),A      ; Ergebnis abspeichern
```

Hier lautet das Objekt-Programm:

```
Adresse  Objekt-Code  Marke   Anweisung
0000                  OP1:    DEFS    1
0001                  OP2:    DEFS    1
0002                  ERG:    DEFS    1
0003     21 00 00             LD      HL,OP1
0006     7E                   LD      A,(HL)
0007     23                   INC     HL
0008     96                   SUB     (HL)
0009     23                   INC     HL
000A     77                   LD      (HL),A
```

Das Programmstück belegt nur 8 Bytes und kann durch Abändern des ersten Befehls auf andere Datenbereiche mit gleicher Struktur angepaßt werden. Läßt man den ersten Befehl weg, so entsteht ein Programmstück, das auf alle Probleme mit obiger Speicherstruktur angewendet werden kann; die jeweilige Adresse des ersten Operanden muß vor Ausführung des Programmstücks ins HL-Register gebracht werden.

Die Laufzeit verringert sich von 47-Takt-Zyklen auf 43-Takt-Zyklen. Besonders ärgerlich beim ersten Programmstück ist die Umstellung der Reihenfolge, in der die Operanden bearbeitet werden müssen; dies kann sich besonders bei aufeinanderfolgenden Operationen bemerkbar machen, bei denen das Ergebnis einer Operation erster Operand der nächsten Operation

ist. Auch der Aufwand zur Dokumentierung ist höher. Bei der zweiten Version stört lediglich, daß es von der oben gezeigten Speicherstruktur abhängig ist.

Übungen

1. Schreibe ein Programmstück, das die Summe von drei hintereinander im Speicher stehenden vorzeichenlosen ganzen 8-Bit-Zahlen bildet und das Ergebnis ohne Berücksichtigung eines Übertrages unmittelbar hinter den drei Zahlen wieder als Byte ablegt. Die Speicheradresse der ersten Zahl soll dabei im HL-Register stehen.

2. Schreibe ein Programmstück, das die Summe von zwei im Speicher stehenden ganzen Zahlen in 2-Komplement-Darstellung mit 8 Bits bildet und das Ergebnis vor den beiden Zahlen als Byte (ohne Berücksichtigung eines Überlaufs) ablegt. Die Speicheradresse der ersten Zahl soll dabei im HL-Register stehen.

3. Im Speicher sollen zwei Folgen von je fünf vorzeichenlosen ganzen 8-Bit-Zahlen stehen, die jeweils fortlaufend abgespeichert sind. Die beiden Folgen sollen komponentenweise addiert werden; die fünf Resultate sollen ebenfalls als Folge von Bytes lückenlos gespeichert werden. Die drei Speicherbereiche liegen dabei nicht unbedingt beieinander. Schreibe ein Programmstück für dieses Problem, das mittels indirekter Adressierung arbeitet.

4. Ein Datenelement vom Typ »Wort« soll im Speicher um 30 Bytes nach hinten verschoben werden. Die Speicheradresse des Worts soll dabei in einem Adreß-Register stehen.

12
Bit-Manipulationen

Bei einer Bit-Manipulation werden ein oder mehrere Bits einer größeren Einheit, zum Beispiel eines Bytes oder Wortes, unabhängig von den übrigen Bits untersucht oder verändert. Bit-Manipulationen im weiteren Sinne sind auch solche Operationen, bei denen der neue Wert eines Bits von den alten oder neuen Werten benachbarter Bits abhängt (Rotations- und Schiebe-Operationen).

12.1 Untersuchung einzelner Bits

Wir wollen annehmen, daß im DE-Register eine ganze Zahl in 2-Komplement-Darstellung steht. Um herauszufinden, ob die Zahl negativ ist, untersuchen wir nur das höchste Bit, da dieses das Vorzeichen angibt. Zur Durchführung des Tests (der übrigens auch für die anderen Komplement-Darstellungen funktioniert) bedienen wir uns des Befehls BIT (bit):

```
BIT     7,D         ; hoechstes Bit des
                    ; DE-Registers testen
JP      NZ,NEGAT    ; Bit 7 gesetzt,
                    ; bedeutet negative Zahl
```

Die Marke NEGAT ist im übrigen Programm geeignet zu definieren.

Der BIT-Befehl bringt das 1-Komplement des untersuchten Bits ins Null-Flag. Ein gesetztes Bit führt also zu einem rückgesetzten Null-Flag, ein rückgesetztes Bit zu einem gesetzten Null-Flag.

Natürlich hätten wir den Test auch mit einem der uns bekannten arithmetischen Befehle durchführen können; dazu müßte aber der Inhalt des D-Registers beziehungsweise eine geeignete Prüfgröße erst ins A-Register gebracht werden, wodurch dieses wiederum zerstört

würde. Der BIT-Befehl hat gegenüber den arithmetischen Befehlen den Vorteil, daß er auf jedes beliebige Bit eines der Register A, B, C, D, E, H, L angewandt werden kann, und daß obendrein das untersuchte Register dabei nicht verändert wird.

Der gleiche Test kann auch auf eine im Speicher stehende Größe vom Typ »Byte« angewandt werden, wenn diese über das HL-Register indirekt adressiert wird. Wollen wir zum Beispiel prüfen, ob die betreffende Größe (als nichtnegative ganze Zahl interpretiert) gerade ist, so schreiben wir:

```
BIT     0,(HL)      ; letztes Bit der
                    ; Zahl untersuchen
JP      Z,GERADE    ; Bit 0 rueckgesetzt,
                    ; also Zahl gerade
```

Auch den schon bekannten Test, ob ein ASCII-codierter Buchstabe groß oder klein ist, können wir mit dem BIT-Befehl effizient durchführen. Wir benutzen dabei, daß Groß- und Kleinbuchstaben sich genau durch den Wert von Bit 5 unterscheiden (das Zeichen soll diesmal im C-Register stehen):

```
BIT     5,C         ; signifikantes Bit des
                    ; Buchstaben untersuchen
JP      NZ,KLEIN    ; Bit 5 gesetzt,
                    ; also Kleinbuchstabe
```

Übungen

1. Schreibe eine Verzweigung, die eine vorzeichenlose ganze Zahl im E-Register genau dann um 1 vermindert, wenn diese ungerade ist.

2. Im H-Register stehe ein ASCII-Buchstabe. Schreibe ein Programmstück, das 0 ins D-Register bringt, wenn der Buchstabe klein ist, −1 dagegen, wenn er groß ist.

3. Schreibe eine Verzweigung, mit der festgestellt werden kann, ob eine durch das HL-Register indirekt adressierte ganze Zahl positiv ist.

12.2 Setzen und Rücksetzen einzelner Bits

Wie wir im vorhergehenden Beispiel gesehen haben, unterscheiden sich ASCII-codierte Großbuchstaben nur durch den Wert von Bit 5 von entsprechenden Kleinbuchstaben. Um aus einem Buchstaben den entsprechenden Kleinbuchstaben zu erhalten, brauchen wir also nur das Bit 5 zu setzen. Dies tun wir mit Hilfe des Befehls **SET** (set), wobei wir den Buchstaben wieder im C-Register erwarten:

```
SET     5,C     ; ASCII-codierten Buchstaben
                ; in Kleinbuchstaben umwandeln
```

Im Gegensatz zu den Lösungen im Kapitel »Verzweigungen« benötigen wir hier keine Verzweigung.

Ebenso einfach erhalten wir einen Großbuchstaben durch Rücksetzen von Bit 5 mittels des Befehls **RES** (reset):

```
RES     5,C     ; ASCII-codierten Buchstaben
                ; in Grossbuchstaben umwandeln
```

Wie beim BIT-Befehl kann auch beim SET-Befehl und beim RES-Befehl ein beliebiges 8-Bit-Hauptregister (außer dem Flag-Register) oder eine durch das HL-Register indirekt adressierte Speicherzelle angegeben werden. Wir sehen uns als Beispiel an, wie der absolute Betrag einer im Speicher stehenden ganzen Zahl in Vorzeichen/Betrag-Darstellung gebildet wird, deren Speicheradresse im HL-Register steht:

```
RES     7,(HL)  ; Vorzeichen positiv machen
```

Steht im L-Register eine ganze Zahl in 2-Komplement-Darstellung, so erhalten wir für gerade Zahlen die nächstgrößere ungerade Zahl durch folgenden Befehl:

```
SET     0,L     ; naechstgroessere
                ; ungerade Zahl erzeugen
```

Übungen

1. Schreibe ein Programm, das mit Hilfe von Bit-Manipulations-Befehlen im Speicher stehende ASCII-codierte Groß- und Kleinbuchstaben vertauscht.

2. Schreibe ein Programm, das eine im B-Register befindliche ganze Zahl in Vorzeichen-/Betrag-Darstellung negiert.

12.3 Speichern von Zuständen

Wir werden manchmal in die Situation kommen, einen Test durchführen zu müssen, dessen Ergebnis wiederholt benötigt wird. Da die Flags durch viele Befehle verändert werden, müssen wir den Zustand eines oder mehrerer Flags irgendwie speichern. Eine Methode ist, durch bestimmte Bits in einem bestimmten Register den Zustand der uns interessierenden Flags wiederzugeben. Nehmen wir zum Beispiel an, wir benötigen den Zustand des Vorzeichen-Flags und wollen diesen in Bit 4 des E-Registers speichern. Ein typisches Programmstück dazu könnte folgendermaßen aussehen:

```
; Zustand des Vorzeichen-Flags in Bit 4 des E-Registers speichern

            SET     4,E         ; Zustand mit 1 vorbesetzen
            JP      M,FERTIG    ; Zustand ist richtig vorbesetzt
            RES     4,E         ; Zustand mit 0 besetzen
FERTIG:     NOP                 ; gemeinsame Fortsetzungsstelle
            :
            :
            :

; Zustand des gespeicherten Flags benutzen

            BIT     4,E         ; Zustand testen
            JP      NZ,NEGAT    ; Vorzeichen-Flag war gesetzt
                                ; bei urspruenglichem Test
```

Es soll an dieser Stelle nochmals darauf hingewiesen werden, daß in einem kompletten Assemblerprogramm der NOP-Befehl normalerweise durch den Befehl ersetzt wird, der unmittelbar nach dem ersten Programmstück ausgeführt werden soll.

Übungen

1. Konstruiere ein Programmstück, in welchem der Zustand des Übertrag-Flags in einem Bit des D-Registers gespeichert wird.

2. Konstruiere ein Programmstück, in welchem der Zustand des Null-Flags in Bit 6 einer durch das HL-Register indirekt adressierten Speicherzelle aufbewahrt wird.

12.4 Maskieren

Eine Größe vom Typ »Byte« kann man durch zwei Hex-Ziffern darstellen; dabei entspricht die niederwertige Hex-Ziffer den Bits 0 bis 3, die höherwertige den Bits 4 bis 7. Wir haben es dabei also mit »Halb-Bytes« zu tun; im Englischen nennt man ein Halb-Byte meist »Nibble«.

Nehmen wir an, daß wir am niederwertigen Nibble eines Bytes interessiert sind. Wir müssen diesen dann vom höherwertigen Nibble isolieren. Dazu bringen wir das Byte erst einmal in das A-Register. Dann löschen wir die Bits 4 bis 7. Wir können dies im Prinzip mit RES-Befehlen tun:

```
            RES     4,A         ; Bit 4 loeschen
            RES     5,A         ; Bit 5 loeschen
            RES     6,A         ; Bit 6 loeschen
            RES     7,A         ; Bit 7 loeschen
```

Da wir mehrere Bits gleichzeitig verändern wollen, bedienen wir uns aber besser einer Technik, die »Maskieren« genannt wird. Dabei verknüpft man den Inhalt des A-Registers bitweise mit einem Datenwert vom Typ »Byte«, der hier die spezielle Funktion einer »Maske« besitzt; die Maske wird oft in binärer Schreibweise angegeben, um ihre Funktion klarer darzustellen. Als Verknüpfungen stehen Konjunktion (AND, Und-Verknüpfung), Disjunktion (OR, Oder-Verknüpfung) und exklusive Disjunktion (XOR, Entweder/Oder-Verknüpfung) zur Verfügung. Die Verknüpfungstabelle für ein Bit sieht dabei so aus:

Tabelle 12.1. *Verknüpfungstabelle für logische Operationen*

1. Operand	2. Operand	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Merke: Die logischen Befehle haben stets implizit das A-Register als ersten Operanden, das Ergebnis der Operation steht immer im A-Register!

Wir lösen unser Problem nun durch den Befehl **AND** (and):

```
AND     00001111B ; Bits 4 bis 7 wegmaskieren
```

Die Maske für einen AND-Befehl konstruieren wir stets nach folgender Methode: Soll ein bestimmtes Bit gelöscht werden, so steht in der Maske an dieser Stelle eine Null; soll der Wert des Bits erhalten bleiben, so steht in der Maske eine Eins.

Dies bedeutet insbesondere, daß man mit Hilfe eines AND-Befehls und einer geeigneten Maske die Befehle RES bit,A und BIT bit,A ersetzen kann. Um den RES-Befehl zu ersetzen, bauen wir eine Maske auf, in der alle Bits bis auf das rückzusetzende den Wert Eins erhalten. Für den BIT-Befehl erzeugen wir eine Maske, in der nur das zu testende Bit den Wert Eins hat. Das Ergebnis der logischen Operation ist damit genau dann Null, wenn das zu testende Bit den Wert Null hat; da auch genau in diesem Fall das Null-Flag gesetzt wird, ist die beschriebene Methode voll kompatibel zur Verwendung des BIT-Befehls.

Der Vorteil, den wir uns bei der Substitution von RES- und BIT-Befehlen durch AND-Befehle verschaffen, besteht darin, daß der zweite Operand des AND-Befehls auch eines der 8-Bit-Hauptregister (außer dem Flag-Register) oder eine durch das HL-Register indirekt adressierte Speicherzelle sein kann. Diese als Masken dienenden Datenwerte – und damit die aktuellen Bitnummern – können im Programm berechnet werden, während dagegen die Bitnummern in RES- und BIT-Befehlen bereits bei der Assemblierung des Programms festliegen und nur durch Änderung des Programms selbst verändert werden können (Finger weg von selbstmodifizierenden Programmen!!!).

Zu diesen Substitutionen eben noch ein Beispiel: wir realisieren die schon hinreichend bekannte Umwandlung von im A-Register stehenden ASCII-Buchstaben in Großbuchstaben durch folgenden Befehl:

```
AND     11011111B ; Buchstaben in
                  ; Grossbuchstaben umwandeln
```

Das Setzen mehrerer Bits gelingt mit dem Befehl **OR** (or), der ganz analog zum AND-Befehl verwendet wird. In der Maske steht dabei eine Eins, falls das entsprechende Bit gesetzt werden soll, eine Null, falls der Wert des Bits erhalten bleiben soll. Wollen wir beispielsweise die Bits 4 und 5 des A-Registers setzen, so schreiben wir

```
OR      00110000B ; Bits 4 und 5
                  ; des A-Registers setzen
```

Bei genauem Hinsehen erkennen wir, daß dies die Umwandlung binär codierter Dezimalziffern in ihre ASCII-Darstellung bewirkt.

Wie der AND-Befehl für RES- und BIT-Befehle, so kann auch der OR-Befehl als Ersatz für den SET-Befehl benutzt werden. In der Maske trägt dabei genau das Bit den Wert Eins, das gesetzt werden soll.

Das Invertieren von Bits geschieht mit dem Befehl **XOR** (exclusive or). Die Maske enthält für jedes Bit, das invertiert werden soll, eine Eins, für jedes Bit, dessen Wert erhalten bleiben soll, eine Null. Zum Beispiel invertieren wir das Bit 7 des A-Registers durch den Befehl

```
XOR     10000000B ; Bit 7 des A-Registers
                  ; invertieren
```

und realisieren damit die Negation für ganze Zahlen in Vorzeichen-/Betrag-Darstellung.

Aus der Verknüpfungstabelle der logischen Operationen ersehen wir, daß die Operationen symmetrisch bezüglich der Reihenfolge der Operanden sind. Die Maske kann somit auch im A-Register stehen, während der zweite Operand den zu maskierenden Wert liefert (als direkten Datenwert, Registerinhalt oder Speicherinhalt); das Ergebnis der Operation steht jedoch stets im A-Register.

Für die simultane Invertierung aller Bits des A-Registers (1-Komplement) gibt es noch den Befehl **CPL** (complement):

```
CPL               ; 1-Komplement des A-Registers
```

Ein beliebter Programmiertrick besteht darin, statt des Befehls LD A,0 den Befehl XOR A zu verwenden, da dieser einen um ein Byte kürzeren Objekt-Code besitzt. Die beiden Befehle unterscheiden sich aber bezüglich der Flags in ihrer Wirkung: Während der LD-Befehl die Flags nicht verändert, werden bei allen logischen Befehlen die Flags folgendermaßen gesetzt:

S:	gesetzt, falls Bit 7 des Ergebnisses gesetzt
Z:	gesetzt, falls Ergebnis Null ist
H:	gesetzt für AND, rückgesetzt für OR und XOR
P:	gesetzt, falls Parität des Ergebnisses gerade ist
N:	rückgesetzt
C:	rückgesetzt

Die Parität einer Bitfolge ist die Anzahl der darin vorkommenden Bits mit dem Wert 1. Die Parität kann man zur Datensicherung bei der Übertragung von Daten ausnutzen (mehr darüber im Kapitel »Ein-/Ausgabe-Techniken«). Aus der Doppelfunktion des P-Flags als Paritäts-Flag und als Überlauf-Flag erklären sich nun auch die merkwürdigen Bezeichnungen PO (parity odd) und PE (parity even), die wir bei den bedingten Sprüngen kennengelernt haben.

Ein weiterer Trick besteht darin, statt des Befehls CP 0 einen der Befehle AND A oder OR A zu verwenden. Das Null-Flag und das Vorzeichen-Flag werden dabei wie durch den CP-Befehl gesetzt. Der Objekt-Code ist wieder um ein Byte kürzer, außerdem erkennen wir am Paritäts-Flag zusätzlich die Parität.

Da bei allen logischen Befehlen das Übertrag-Flag rückgesetzt wird, verwendet man statt der uns bereits bekannten Befehlsfolge

```
SCF                    ; Uebertrag-Flag setzen
CCF                    ; Uebertrag-Flag loeschen
```

meist einen der Befehle AND A oder OR A, da diese in bezug auf das Übertrag-Flag genauso wirken, aber einen kürzeren Objekt-Code ergeben. Der Inhalt des A-Registers bleibt dabei erhalten, nicht jedoch die Zustände der übrigen Flags.

Die beschriebenen Programmiertricks sollte man mit Vorsicht verwenden, da die Befehle in ihrer Wirkung eben doch nicht genau überstimmen; am besten schreibt man erst einmal die längeren, aber klareren Befehle ins Programm, um in einem späteren Optimierlauf zu prüfen, ob sie gefahrlos durch die kürzeren ersetzt werden können. Auf jeden Fall müssen solche Optimierungen genau dokumentiert werden!

Überlege Dir sorgfältig, warum die hier angegebenen Optimierungen wirken und welche Nebeneffekte dabei auftreten! Sieh Dir auch einmal die Beschreibung der Befehle im Anhang genau an!

Übungen

1. Wandle durch Maskieren ASCII-codierte Dezimalziffern in ihre Binärdarstellung um.

2. Realisiere den absoluten Betrag für ganze Zahlen in Vorzeichen-/Betrag-Darstellung durch Maskieren.

3. Ersetze durch Maskieren ASCII-Buchstaben durch Kleinbuchstaben.

4. Vertausche Groß- und Kleinbuchstaben durch Maskieren.

5. Vertausche ASCII-Codierung und Binärcodierung von Dezimalziffern durch Maskieren.

6. Schreibe ein Programm, das wechselseitig die Umwandlung von ganzen 8-Bit-Zahlen in Vorzeichen-/Betrag-Darstellung, 1-Komplement-Darstellung und 2-Komplement-Darstellung vornimmt. Codiere dabei die auszuführende Funktion nach Belieben in einem Register.

7. Modifiziere das Programm aus Aufgabe 6 so, daß es für 16-Bit-Zahlen funktioniert.

12.5 Verschieben und Rotieren

Wir wollen als nächstes die Aufgabe lösen, den höherwertigen Nibble eines Bytes zu isolieren. Durch Maskieren kann zwar der niederwertige Nibble entfernt werden, die entstehende Größe vom Typ »Byte« stellt jedoch - als ganze Zahl interpretiert - nicht die Binärcodierung der höherwertigen Hex-Ziffer dar. Wir erreichen unser Ziel, indem wir den Inhalt des A-Registers viermal nach rechts verschieben und dabei links jeweils eine Null nachziehen (Bit 7 soll dabei ganz links stehen, Bit 0 ganz rechts). Eine derartige Operation heißt »logische Rechts-Verschiebung«; sie wird durch den Befehl **SRL** (shift right logically) realisiert. Die logische Rechts-Verschiebung kann man sich folgendermaßen vorstellen:

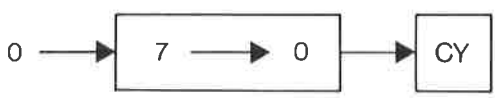


Bild 12.1. *Logische Rechts-Verschiebung (SRL-Befehl)*

Die geforderte Aufgabe lösen wir also durch folgendes Programmstück:

```
SRL     A       ; Inhalt des A-Registers um ein
                ; Bit logisch rechts-verschieben
SRL     A       ; Inhalt des A-Registers um ein
                ; Bit logisch rechts-verschieben
SRL     A       ; Inhalt des A-Registers um ein
                ; Bit logisch rechts-verschieben
SRL     A       ; Inhalt des A-Registers um ein
                ; Bit logisch rechts-verschieben
```

Der Objekt-Code lautet dabei:

Adresse	Objekt-Code	Marke	Anweisung	
0000	CB 3F		SRL	A
0002	CB 3F		SRL	A
0004	CB 3F		SRL	A
0006	CB 3F		SRL	A

Als Nebeneffekt des SRL-Befehls wird der alte Inhalt von Bit 0 ins Carry-Flag übertragen. Als Operand des SRL-Befehls ist jedes der Register A, B, C, D, E, H, L oder eine durch das HL-Register indirekt adressierte Speicherzelle zulässig.

Wenn wir umgekehrt eine binär-codierte Hex-Ziffer in den höherwertigen Nibble eines Bytes bringen wollen, so können wir dazu den Befehl **SLA** (shift left arithmetically) benutzen, der um ein Bit nach links verschiebt, rechts eine Null nachzieht und den alten Inhalt von Bit 7 ins Carry-Flag überträgt. Im Bild:

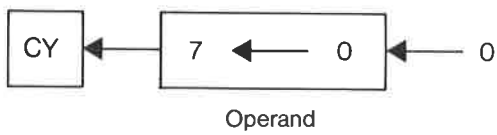


Bild 12.2. *Arithmetische Links-Verschiebung (SLA-Befehl)*

Das entsprechende Programmstück würde also lauten (diesmal wollen wir den Nibble im C-Register aufbauen):

```
SLA     C       ; Inhalt des C-Registers
                ; um ein Bit
                ; arithmetisch links-verschieben
SLA     C       ; Inhalt des C-Registers
                ; um ein Bit
                ; arithmetisch links-verschieben
SLA     C       ; Inhalt des C-Registers
                ; um ein Bit
                ; arithmetisch links-verschieben
SLA     C       ; Inhalt des C-Registers
                ; um ein Bit
                ; arithmetisch links-verschieben
```

Die Befehle SLA A und ADD A,A unterscheiden sich nur bezüglich ihrer Wirkung auf das Überlauf-Flag und das Hilfs-Übertrag-Flag (siehe Anhang A); sind wir am Zustand dieser beiden Flags nicht interessiert, so verwenden wir den kürzeren und schnelleren ADD-Befehl.

Im Kapitel »ganze Zahlen« werden wir die »arithmetische Rechts-Verschiebung« benutzen,

die sich von der logischen Rechts-Verschiebung dadurch unterscheidet, daß der Zustand von Bit 7 stets erhalten bleibt (für eine Begründung siehe das angegebene Kapitel). Schematisch lautet sie:

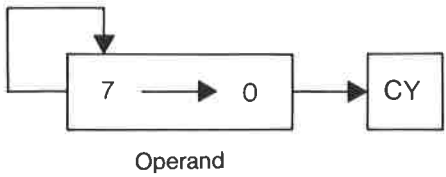


Bild 12.3. *Arithmetische Rechts-Verschiebung (SRA-Befehl)*

Die arithmetische Rechts-Verschiebung wird durch den Befehl **SRA** (shift right arithmetically) realisiert, zum Beispiel für eine indirekt adressierte Speicherzelle:

```
SRA         (HL)        ; Inhalt der
                        ; durch das HL-Register
                        ; indirekt adressierten
                        ; Speicherzelle um ein Bit
                        ; arithmetisch rechts-verschieben
```

Das Zusammensetzen eines Bytes aus den beiden Nibbles erfolgt durch den OR-Befehl (wir nehmen den einen Nibble im A-Register, den anderen im C-Register an):

```
OR          C           ; zwei Nibbles
                        ; zu Byte zusammensetzen
```

Wir hätten auch den Befehl ADD A,C dazu verwenden können. Der logische Befehl unterscheidet sich aber bezüglich der Wirkung auf die Flags; außerdem ist das Zusammensetzen dem Charakter nach keine arithmetische, sondern eine logische Operation.

Mit den beschriebenen Techniken können wir nun beliebige Ausschnitte aus einem Byte isolieren beziehungsweise ein Byte aus verschiedenen Abschnitten zusammensetzen. Wollen wir dasselbe mit einem Wort durchführen, so muß beim Verschieben zwischen den beiden beteiligten 8-Bit-Registern ein Bit übertragen werden. Mit den Verschiebe-Befehlen ist das nicht möglich; wir bedienen uns deshalb der sogenannten Rotations-Befehle, der Austausch eines Bits erfolgt über das Übertrag-Flag.

Der Befehl **RR** (rotate right) verschiebt wie der SRA-Befehl und der SRL-Befehl um ein Bit nach rechts, Bit 7 erhält dabei aber den alten Wert des Übertrag-Flags. Die Rechts-Rotation sieht damit folgendermaßen aus:

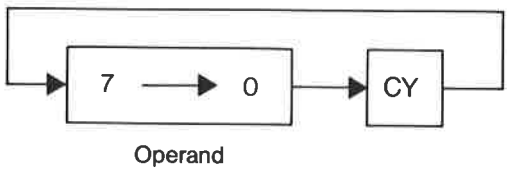


Bild 12.4. *Rechts-Rotation (RR-Befehl)*

Eine logische Rechts-Verschiebung des DE-Registers würde damit so aussehen:

```
SRL     D       ; D-Register
                ; logisch rechts-verschieben
RR      E       ; E-Register rechts-verschieben,
                ; dabei Bit 0 von D-Register
                ; uebernehmen
```

Genau betrachtet rotieren wir mit dem RR-Befehl eigentlich nicht das betreffende Register, sondern eine Konkatenation des Übertrag-Flags (als 1-Bit-Register interpretiert) mit dem Register. Wir können dies formal durch den Konkatenations-Operator & darstellen, in der formalen Beschreibungssprache lautet unser Beispiel:

D & CY <– 0 & <D>
E & CY <– <CY> & <E>

D & CY steht hier für die Konkatenation von $D_7, D_6 \ldots D_0$, CY. Entsprechendes gilt für die anderen Register.

Eine arithmetische Rechts-Verschiebung des HL-Registers realisieren wir durch

```
SRA     H       ; H-Register arithmetisch
                ; rechts-verschieben
RR      L       ; L-Register rechts-verschieben,
                ; dabei Bit 0
                ; von H-Register uebernehmen
```

In der Beschreibungssprache entspricht dem

H & CY <– <H_7> & <H>
L & CY <– <CY> & <L>

Die Links-Verschiebung eines Doppelregisters erhalten wir mit Hilfe des Befehls **RL** (rotate left), der eine Links-Rotation der Konkatenation aus Carry-Flag und Register bewirkt. Als Bild dargestellt:

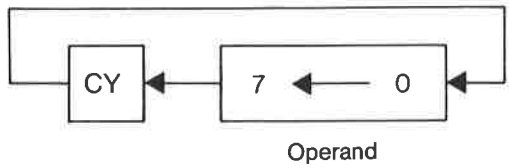


Bild 12.5. *Links-Rotation (RL-Befehl)*

Zum Beispiel für das BC-Register:

```
SLA     C       ; C-Register links-verschieben
RL      B       ; B-Register links-verschieben,
                ; dabei Bit 7
                ; von C-Register uebernehmen
```

In der Beschreibungssprache ausgedrückt:

CY & C <– <C> & 0
CY & B <– <B> & <CY>

Achte darauf, in welcher Reihenfolge die Register bearbeitet werden, und wie Verschieben und Rotieren ineinander greift.

Die Links-Verschiebung des HL-Registers führt man meist mit dem kürzeren und schnelleren Befehl ADD HL,HL durch, der aber die Flags anders setzt.

Wollen wir statt der Konkatenation des Übertrag-Flags mit einem Register das Register allein nach rechts rotieren, so benutzen wir den Befehl RRC (rotate right circular); der alte Wert von Bit 0 wird dabei ins Übertrag-Flag kopiert. Bildlich dargestellt:

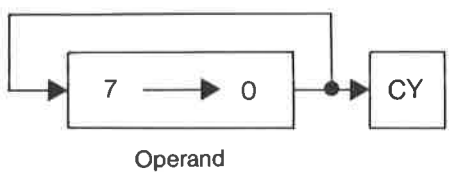


Bild 12.6. *Zirkuläre Rechts-Rotation (RRC-Befehl)*

Beispielsweise für das E-Register:

```
RRC     E       ; E-Register zirkulaer
                ; rechts-rotieren
```

In der Beschreibungssprache lautet diese Operation:

E & CY <– <E_0> & <E>

Entsprechend gibt es für die zirkuläre Links-Rotation den Befehl **RLC** (rotate left circular), der den alten Wert von Bit 7 ins Übertrag-Flag bringt. Bildlich ausgedrückt:

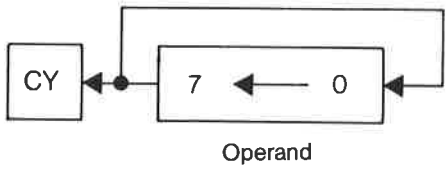


Bild 12.7. *Zirkuläre Links-Rotation (RLC-Befehl)*

Am Beispiel des H-Registers:

```
RLC         H          ; H-Register zyklisch
                       ; links rotieren
```

In der Beschreibungssprache ausgedrückt:

CY & H <– <H> & <H_7>

Wie bei den Verschiebe-Befehlen kann auch bei den Rotations-Befehlen statt eines 8-Bit-Hauptregisters (außer Flag-Register) eine durch das HL-Register indirekt adressierte Speicherzelle angegeben werden.

Zum Rotieren des A-Registers gibt es vier weitere Befehle, die im Prinzip durch die bereits besprochenen Befehle realisiert werden könnten, die sich aber durch einen kürzeren Objekt-Code und eine kürzere Ausführungszeit sowie durch eine andere Behandlung der Flags von diesen unterscheiden. Es sind dies:

RLA	statt	RL A
RRA	statt	RR A
RLCA	statt	RLC A
RRCA	statt	RRC A

Mit Hilfe dieser Befehle können wir das erste Beispiel dieses Unterkapitels – das Isolieren und Rechts-Verschieben des höherwertigen Nibbles eines Bytes – noch effizienter lösen:

```
AND         11110000B  ; niederwertigen Nibble
                       ; ausblenden
RRCA                   ; hoeherwertigen Nibble
                       ; rechts-verschieben
RRCA                   ; hoeherwertigen Nibble
                       ; rechts-verschieben
```

```
        RRCA                    ; hoeherwertigen Nibble
                                ; rechts-verschieben
        RRCA                    ; hoeherwertigen Nibble
                                ; rechts-verschieben
```

Als Objekt-Code erhalten wir nun:

Adresse	Objekt-Code	Marke	Anweisung	
0000	E6 F0		AND	11110000B
0002	0F		RRCA	
0003	0F		RRCA	
0004	0F		RRCA	
0005	0F		RRCA	

Sehen Sie sich dieses Beispiel ganz genau an und lesen Sie erst weiter, wenn Sie es auch sicher verstanden haben!

Weil das Arbeiten mit Nibbles so häufig vorkommt (insbesondere deswegen, weil man mit einem Nibble eine Dezimalziffer codieren kann, sogenannte BCD-Codierung), besitzt der Z80 zwei spezielle Befehle für das Rotieren von Nibbles. Dies sind die beiden Befehle **RLD** (rotate left digit) und **RRD** (rotate right digit). Beide stellen eine zirkuläre Rotation um vier Bits auf dem Superregister A_3 & A_2 & A_1 & A_0 & (<HL>) dar.

Der RLD-Befehl rotiert dieses Superregister zirkulär nach links, also

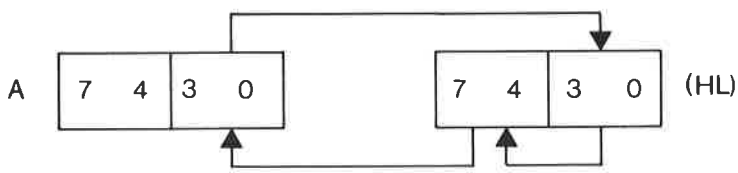


Bild 12.8. *Zirkuläre Links-Rotation von Nibbles (RLD-Befehl)*

In der Beschreibungssprache lautet dies:

A_3 & A_2 & A_1 & A_0 & (<HL>) <– <(<HL>)> & <A_3> & <A_2> & <A_1> & <A_0>

Der RRD-Befehl rotiert dagegen zirkulär nach rechts:

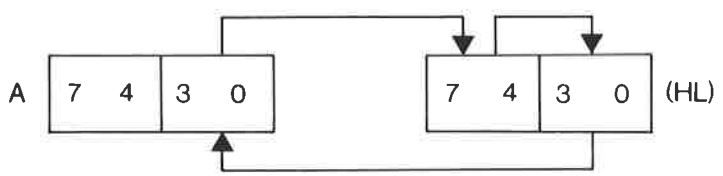


Bild 12.9. *Zirkuläre Rechts-Rotation von Nibbles (RRD-Befehl)*

Formal beschrieben:

(<HL>) & A_3 & A_2 & A_1 & A_0 <– <A_3> & <A_2> & <A_1> & <A_0> & <(<HL>)>

Wir werden diese beiden Befehle intensiv in den Kapiteln »Ganze Zahlen« und »Gleitpunkt-Zahlen« benutzen.

Übungen

1. Verwandle ein im A-Register stehendes Byte in zwei Nibbles und lege diese als ASCII-codierte Hex-Ziffern im B-Register und C-Register ab.

2. In den Registern D und E stehe je eine ASCII-codierte Hex-Ziffer. Setze im A-Register das Byte zusammen, dessen höherwertiger Nibble im D-Register, und dessen niederwertiger Nibble im E-Register codiert ist.

3. Wir fassen die Register B, C, D und E als 32-Bit-Superregister auf, das sich durch die Konkatenation B & C & D & E ergibt. Schreibe je ein Programm für die arithmetische Rechts-Verschiebung, arithmetische Links-Verschiebung und logische Rechts-Verschiebung dieses Superregisters.

4. Modifiziere die Programme aus Aufgabe 1 und 2 so, daß statt des A-Registers eine durch das HL-Register indirekt adressierte Speicherzelle benutzt wird.

13
Schleifen

Schleifen sind Konstrukte zur wiederholten Abarbeitung von Programmstücken. Eine Schleife besteht stets aus zwei Teilen: der Schleifenkörper ist das zu wiederholende Programmstück, die Schleifensteuerung sorgt für den korrekten Ablauf des Vorgangs.

Wird bei Eintritt in die Schleife geprüft, ob der Schleifenkörper überhaupt ausgeführt werden soll, so spricht man von einer abweisenden Schleife; bei einer annehmenden Schleife wird der Schleifenkörper dagegen stets mindestens einmal ausgeführt. Außerdem kann eine Schleife auch vor dem regulären Ende der Abarbeitung verlassen werden; dies nennt man einen Abbruch. Es gibt auch endlose Schleifen, das sind solche ohne Ende-Kriterium. Endlose Schleifen können nur durch Abbruch verlassen werden.

Eine Zählschleife ist eine Schleife mit einer fest vorgegebenen Anzahl von Durchläufen. Die Anzahl der Durchläufe steuert eine Zählgröße; wird diese nach jedem Durchlauf vermindert, so spricht man von absteigender Zählung, ansonsten von aufsteigender Zählung. Für die automatische Steuerung einer annehmenden absteigenden Zählschleife mit dem B-Register als Zählgröße besitzt der Z80 einen speziellen Befehl; alle anderen Formen von Schleifen muß der Programmierer selbst steuern.

13.1 Die automatische Zählschleife

Die automatische Zählschleife des Z80 realisiert man mit Hilfe des Befehls **DJNZ** (decrement and jump if not zero). Als Zählgröße dient das B-Register. Der DJNZ-Befehl dekrementiert (das bedeutet: vermindert um 1) als erstes den Inhalt des B-Registers. Falls der neue Inhalt von Null verschieden ist, erfolgt ein Sprung auf den Anfang des Schleifenkörpers; der Sprung ist als relativer Sprung wie im JR-Befehl ausgelegt. Die Sprungdistanz ist der einzige explizite Operand des DJNZ-Befehls. Wie im JR-Befehl kann statt der Sprungdistanz einfach eine Marke angegeben werden, die am Anfang des Schleifenkörpers definiert ist. Die Intention der automatischen Zählschleife kann formal folgendermaßen zum Ausdruck gebracht werden:

wiederhole
Schleifenkörper
mit B-Register von Startwert **bis** 1 **in Schritten von** −1

Der DJNZ-Befehl wird unmittelbar nach Abarbeitung des Schleifenkörpers ausgeführt.

Der Startwert muß vor Ausführung der Schleife ins B-Register gebracht werden. Die automatische Zählschleife ist eine annehmende Schleife, was zur Folge hat, daß der Startwert 0 als Startwert 256 interpretiert wird (nach dem ersten Abarbeiten des Schleifenkörpers wird das B-Register dann auf 255 dekrementiert). Die Zählgröße kann auch im Schleifenkörper verändert werden. Diese Technik ist bei der automatischen Zählschleife aber häufig ein Zeichen von schlechtem Programmierstil; trotzdem sollten wir die automatische Zählschleife korrekt so beschreiben:

wiederhole
Schleifenkörper
B <– <B> – 1
bis <B> = 0

Als erstes Beispiel wollen wir eine (allerdings ineffiziente) Multiplikationsroutine programmieren. Im B-Register und im C-Register stehe dazu je eine vorzeichenlose ganze Zahl. Diese beiden Zahlen sollen miteinander multipliziert werden, das Ergebnis soll ins HL-Register gebracht werden. Wir lösen das Problem durch sukzessive Addition. Als erstes bereiten wir die Register für die Addition vor; dann folgt die eigentliche Schleife. Dabei beachten wir, daß der Multiplikator 0 das Ergebnis 0 liefern muß. Der Algorithmus lautet formal:

HL <– 0
wenn <B> ungleich 0
dann DE <– <C>
wiederhole
HL <– <HL> + <DE>
mit B-Register von <B> bis 1 in Schritten von −1

Im Flußdiagramm ausgedrückt, Bild 13.1.

Das zugehörige Programm lautet:

```
LD      HL,0        ; Akkumulator loeschen
INC     B           ; Multiplikator auf
DEC     B           ; Inhalt Null testen
JP      Z,FERTIG    ; Multiplikator ist Null,
                    ; nichts zu tun
LD      E,C         ; Multiplikand
LD      D,H         ; D <– 0, das heisst
```

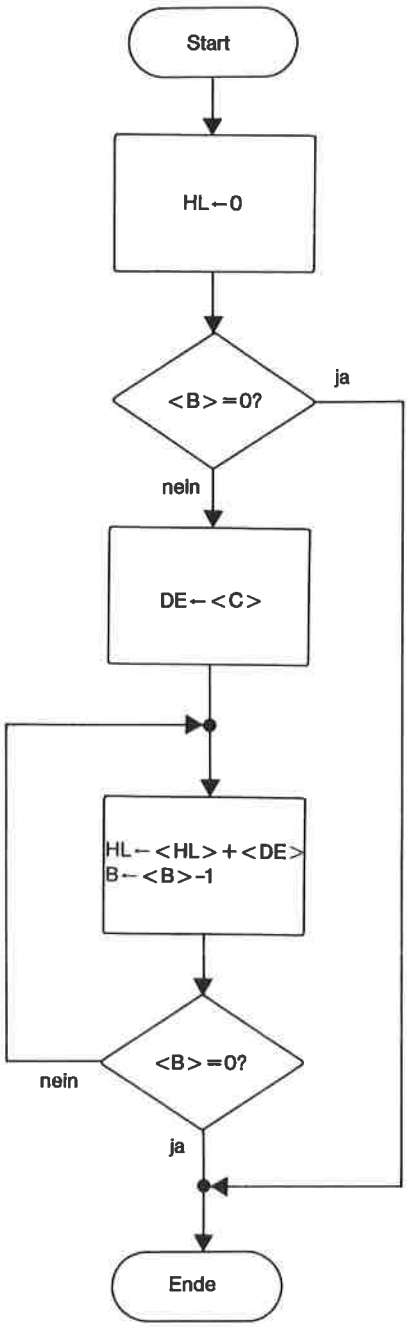


Bild 13.1. *Flußdiagramm: Einfache Multiplikation*

```
                                    ; Multiplikand zu
                                    ; 16-Bit-Groesse erweitern
MULTI:  ADD     HL,DE       ; Multiplikand zu Akkumulator
                            ; addieren
        DJNZ    MULTI       ; Multiplikator abwickeln
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Als Objekt-Code erhalten wir:

Adresse	Objekt-Code	Marke	Anweisung	
0000	21 00 00		LD	HL,0
0003	04		INC	B
0004	05		DEC	B
0005	CA 0D 00		JP	Z,FERTIG
0008	59		LD	E,C
0009	54		LD	D,H
000A	19	MULTI:	ADD	HL,DE
000B	10 FD		DJNZ	MULTI
000D	00	FERTIG:	NOP	

Die Befehle **INC** (increment) und **DEC** (decrement) für 8-Bit-Register entsprechen in diesem Beispiel den bereits bekannten INC- und DEC-Befehlen für 16-Bit-Register. Allerdings beeinflussen sie die Flags, wie Arithmetik-Befehle dies auch tun würden. Die aufeinanderfolgende Ausführung der Befehle INC und DEC für einen 8-Bit-Operanden ist ein beliebter Trick, um das Null-Flag oder das Vorzeichen-Flag entsprechend dem Inhalt eines Registers oder einer Speicherzelle zu setzen, ohne das A-Register zu benutzen; der Inhalt der angesprochenen Größe bleibt bei dieser Operationsfolge erhalten, die sich im übrigen durch effizienten Objekt-Code auszeichnet.

Die 8-Bit-INC- und DEC-Befehle gibt es für die Register A, B, C, D, E, H, L sowie für eine durch das HL-Register indirekt adressierte Speicherzelle; sie setzen die Flags folgendermaßen:

S:	gesetzt, falls Resultat negativ ist
Z:	gesetzt, falls Resultat Null ist
H:	gesetzt, falls Übertrag von Bit 3
P:	gesetzt, falls Überlauf auftrat
N:	rückgesetzt bei INC, gesetzt bei DEC
C:	unverändert

Das B-Register kann in der automatischen Zählschleife als Parameter für den Schleifenkörper dienen. Wir zeigen dazu, wie zu einer im B-Register stehenden positiven ganzen Zahl n die Summe der Zahlen 1 bis n gebildet werden kann. Zuerst die Darstellung des Algorithmus als Flußdiagramm:

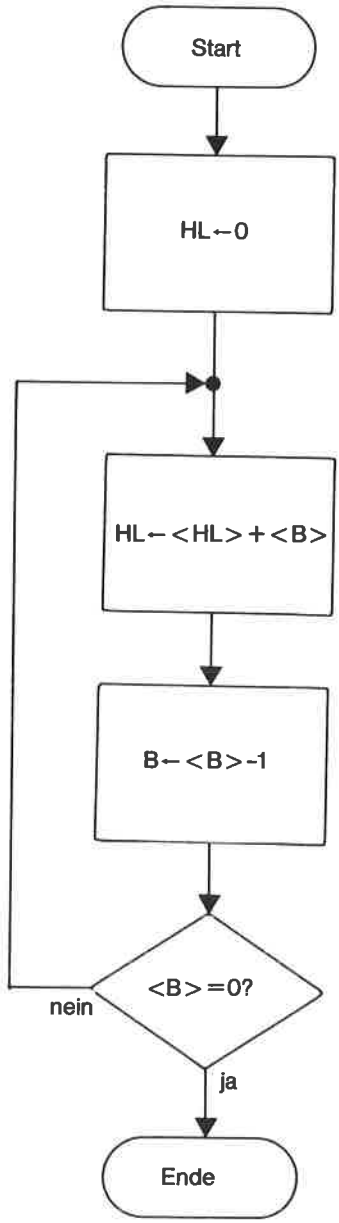


Bild 13.2. *Flußdiagramm: Berechnung der Summe von 1 bis n*

Das zugehörige Programm lautet:

```
        LD      HL,0        ; Akkumulator loeschen
        LD      D,H         ; D <- 0: Vorbereitung, um
                            ; B-Register zu einer 16-Bit-
                            ; Groesse zu erweitern
SUMME:  LD      E,B         ; Summand ins DE-Register bringen
        ADD     HL,DE       ; Summand zu Akkumulator addieren
        DJNZ    SUMME       ; Zahlen absteigend durchlaufen
```

Durch einen Trick können wir die annehmende Zählschleife auch in eine abweisende Zählschleife umbauen. Dazu springen wir als erstes auf den DJNZ-Befehl, der dann entscheidet, ob die Schleife überhaupt ausgeführt wird. Die Zählgröße B muß dabei natürlich um 1 größer sein als im Falle der annehmenden Schleife, da sie ja vor der ersten Ausführung des Schleifenkörpers bereits einmal dekrementiert wird. Wir schreiben das erste Beispiel dieses Unterkapitels als abweisende Schleife, was zur Folge hat, daß auch die Null als Multiplikator auftreten darf, ohne daß dafür eine Sonderbehandlung erforderlich wäre. Im Flußdiagramm, Bild 13.3.

Als Programm erhalten wir:

```
        LD      HL,0        ; Akkumulator loeschen
        INC     B           ; Zaehlgroesse fuer abweisende
                            ; Schleife um 1 erhoehen
        LD      E,C         ; Multiplikand
        LD      D,H         ; D <- 0, das heisst
                            ; Multiplikand zu
                            ; 16-Bit-Groesse erweitern
        JP      TEST        ; Schleife abweisend machen
MULTI:  ADD     HL,DE       ; Multiplikand zu Akkumulator
                            ; addieren
TEST:   DJNZ    MULTI       ; Multiplikator abwickeln
```

Zum Abschluß präsentieren wir noch eine effiziente Multiplikations-Routine für zwei 8-Bit-Operanden, die als vorzeichenlose ganze Zahlen gedeutet werden. Die Operanden sollen dabei im C-Register und im E-Register angeliefert werden, das Ergebnis fällt im HL-Register an (das Ergebnis benötigt ja eventuell mehr als 8 Bits). Der zugrunde liegende Algorithmus ist im Kapitel »Sequenzen« beschrieben; jedoch sind diesmal beide Operanden variabel. Der Algorithmus als Flußdiagramm siehe Bild 13.4.

Das entsprechende Programm ist als automatische Zählschleife formuliert:

```
        LD      HL,0        ; Akkumulator loeschen
        LD      D,H         ; Multiplikand zu 16-Bit-
                            ; Groesse erweitern
```

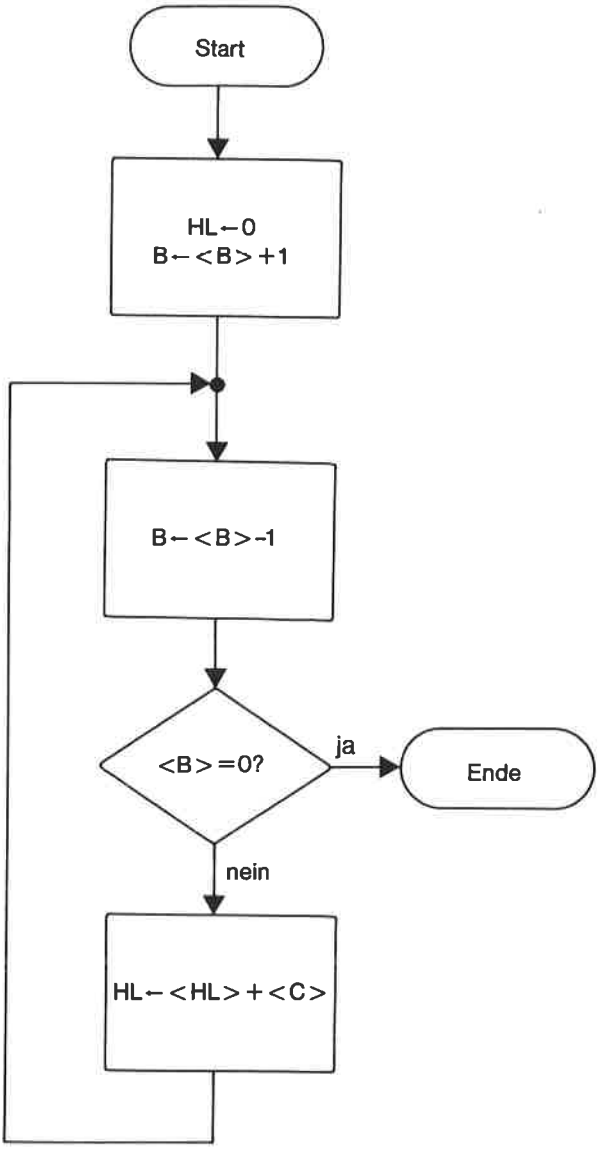


Bild 13.3. *Flußdiagramm: Multiplikation mit abweisender Schleife*

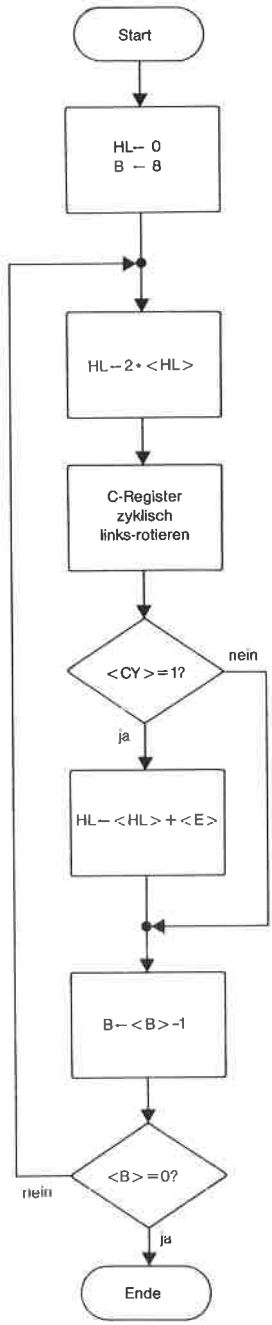


Bild 13.4. *Flußdiagramm: Effiziente Multiplikation*

```
        LD      B,8     ; Schleifenzaehler mit Laenge
                        ; des Multiplikators (in Bits)
                        ; besetzen
MULTI:  ADD     HL,HL   ; Akkumulator verdoppeln
        RLC     C       ; hoechstes Bit des
                        ; Multiplikators ins Uebertrag-
                        ; Flag bringen, gleichzeitig
                        ; Multiplikator links-rotieren
        JP      NC,NULL ; keine Addition erforderlich,
                        ; wenn anstehendes Bit des
                        ; Multiplikators 0 ist
        ADD     HL,DE   ; Multiplikand zu Akkumulator
                        ; addieren
NULL:   DJNZ    MULTI   ; naechstes Bit des
                        ; Multiplikators verarbeiten
```

Das angegebene Programm hat die angenehme Eigenschaft, daß nach Ausführung der Multiplikation beide Operanden ihren ursprünglichen Wert besitzen.

Übungen

1. Schreibe ein Programmstück, das eine Maske zum Setzen von Bit n einer 8-Bit-Größe berechnet (siehe dazu Unterkapitel 12.4). Die Zahl n stehe dabei im C-Register.

2. Schreibe ein Programmstück, das den Inhalt des DE-Registers um n Bits zirkulär rechts rotiert. Die Zahl n stehe dabei im A-Register.

3. Modifiziere das letzte Programm dieses Unterkapitels so, daß zwei ganze Zahlen in 2-Komplement-Darstellung multipliziert werden können.

13.2 Selbstgesteuerte annehmende Zählschleifen

Alle selbstgesteuerten, das heißt vom Programmierer selbst gesteuerten, Schleifen arbeiten mit Sprung-Befehlen. Bei der annehmenden absteigenden Zählschleife wird nach Abarbeitung des Schleifenkörpers eine Zählgröße um einen bestimmten Wert (der im allgemeinen von 1 verschieden ist) vermindert; wird dabei ein vorgegebener Endwert unterschritten, so terminiert die Schleife. Ansonsten erfolgt ein Sprung auf den Anfang des Schleifenkörpers. Die symbolische Form lautet:

wiederhole
        Schleifenkörper
mit Schleifenzähler von Startwert **bis** Endwert **in Schritten von** Schrittweite

Die Schrittweite ist hier negativ, der jeweils nächste Wert des Schleifenzählers ergibt sich durch Addition der Schrittweite zum bisherigen Wert des Schleifenzählers. Der Schleifenzähler kann in einem Register, in der Konkatenation mehrerer Register oder im Speicher stehen.

Wir betrachten folgendes Beispiel: Der Speicherbereich 3000H bis 4FFFH soll mit Nullen überschrieben werden. In allgemeiner Form lautet der Algorithmus:

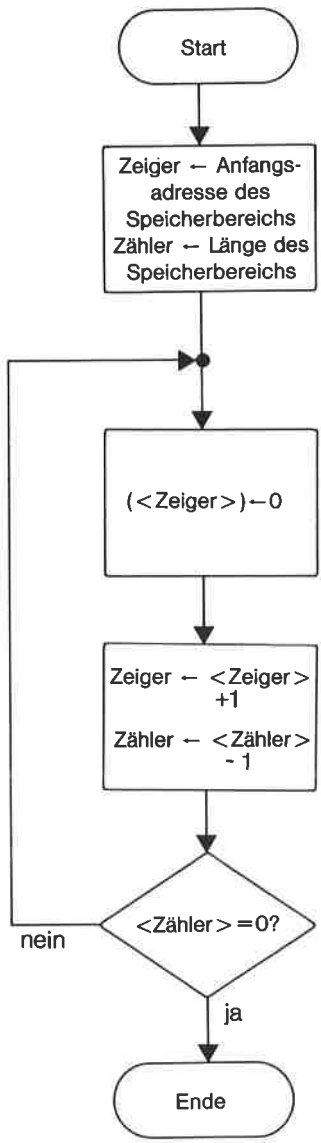


Bild 13.5. *Flußdiagramm: Überschreiben eines Speicherbereichs*

Wir verwenden das BC-Register als Zählgröße, das HL-Register als Adreß-Register:

```
        LD      HL,3000H        ; Zeiger auf Speicherbereich
        LD      BC,5000H-3000H  ; Laenge des Speicherbereichs
FUELL:  LD      (HL),0          ; Speicherzelle fuellen
        INC     HL              ; auf naechste Speicherzelle
                                ; zeigen
        DEC     BC              ; Schrittweite zu Zaehler
                                ; addieren
        LD      A,B             ; Test auf <BC> = 0 vorbereiten
        OR      C               ; das Null-Flag ist jetzt genau
                                ; dann gesetzt, wenn <BC> = 0
        JP      NZ,FUELL        ; Zaehler ungleich Null,
                                ; zum Schleifenanfang springen
```

Wir sehen hier, daß das Endkriterium für 16-Bit-Zählgrößen komplizierter aufgebaut ist als für 8-Bit-Größen. Der angewandte Test auf <BC> = 0 ist eine gebräuchliche Methode für Größen, die sich über mehrere Register verteilen. Wenn wir zum Beispiel prüfen wollen, ob die Register D, E, H, L allesamt Null enthalten, so schreiben wir:

```
        LD      A,D
        OR      E
        OR      H
        OR      L
```

Enthalten alle Register den Wert Null, so auch das A-Register; damit ist auch das Null-Flag gesetzt und kann getestet werden. Ist jedoch der Inhalt mindestens eines Registers von Null verschieden, so ist auch der Inhalt des A-Registers von Null verschieden, das Null-Flag also gelöscht.

Genauso können wir auch eine Folge von Speicherzellen darauf testen, ob alle den Wert Null enthalten. Wollen wir beispielsweise die drei Speicherzellen mit den Adressen 468AH, 468BH und 468CH prüfen, so lautet das entsprechende Programmstück:

```
        LD      HL,468AH        ; Daten-Adress-Register mit
                                ; erster Adresse laden
        LD      A,(HL)          ; Test vorbereiten
        INC     HL              ; auf zweiten Wert zeigen
        OR      (HL)            ; zweiten Wert verknuepfen
        INC     HL              ; auf dritten Wert zeigen
        OR      (HL)            ; dritten Wert verknuepfen
```

Im allgemeinen werden wir auch andere Schrittweiten und Endwerte berücksichtigen müssen. Seien zum Beispiel m und n zwei ganze Zahlen mit $2 < m < n$. Wir wollen nun die Summe der

Zahlen n, n–3, n–6, ... berechnen, wobei keiner der Summanden kleiner als m sein soll. Wir nehmen an, daß m im C-Register steht, n im B-Register. Das A-Register benutzen wir als Zählgröße und bilden die Summe im HL-Register nach folgender Vorschrift:

HL <– 0
wiederhole
HL <– <HL> + <A>
mit A von <B> **bis** <C> **in Schritten von** –3

In allgemeiner Form als Flußdiagramm formuliert, Bild 13.6.

Zur Realisierung der Schrittweite –3 vermindern wir nach jedem Schleifendurchlauf die Zählgröße um 3. Wird diese dabei kleiner als der Endwert, so terminiert die Schleife:

```
        LD      HL,0        ; Akkumulator loeschen
        LD      D,H         ; D <– 0
        LD      A,B         ; Zaehlgroesse mit Startwert
                            ; laden
ADD:    LD      E,A         ; Summand ins DE-Register bringen
        ADD     HL,DE       ; Summand zu Akkumulator addieren
        SUB     3           ; Schrittweite zu Zaehler
                            ; addieren
        CP      C           ; mit Endwert vergleichen
        JP      NC,ADD      ; zum Schleifenanfang springen,
                            ; falls Zaehlgroesse nicht
                            ; kleiner als Endwert
```

Vorsicht bei diesen Tests auf das Ende einer Schleife, es schleichen sich hier leicht Fehler ein! Überlege Dir, was zum Beispiel für die unzulässigen Parameter m=1 und n=4 passieren würde!

Die annehmende aufsteigende Zählschleife unterscheidet sich in der symbolischen Form nicht von der annehmenden absteigenden Zählschleife: Nach Abarbeitung des Schleifenkörpers wird die Zählgröße um die Schrittweite erhöht; wird dabei der Endwert überschritten, so terminiert die Schleife. Die Schrittweite ist hier positiv. Die Realisierung der annehmenden aufsteigenden Zählschleife erfolgt analog zur annehmenden absteigenden Zählschleife. Wir wollen das erste Beispiel dieses Unterkapitels noch einmal aufgreifen und dabei das BC-Register als Adreß-Register und als Zählgröße zugleich verwenden. Zuerst das Flußdiagramm mit dem allgemeinen Verfahren (siehe Bild 13.7.).

Das entsprechende Programm lautet:

```
        LD      BC,3000H    ; Zeiger auf Speicherbereich
FUELL:  XOR     A           ; ein Programmiertrick:
                            ; A-Register mit Null laden
                            ; und Uebertrag-Flag loeschen
```

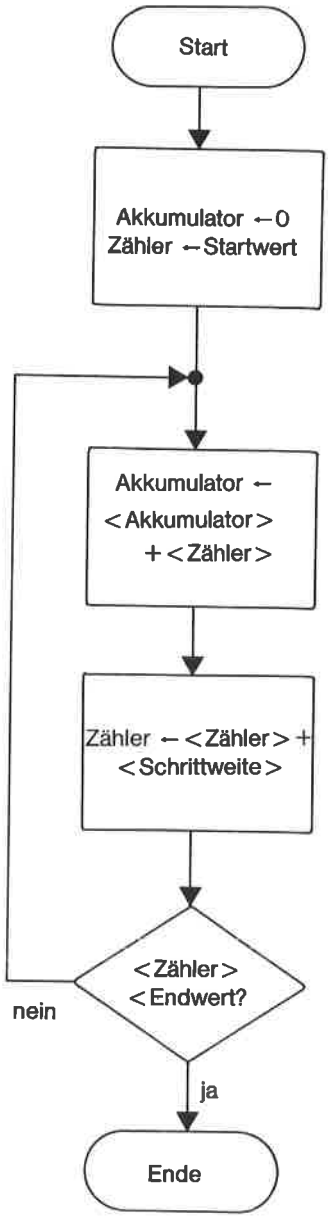


Bild 13.6. *Flußdiagramm: Summenbildung mit absteigender Zählschleife*

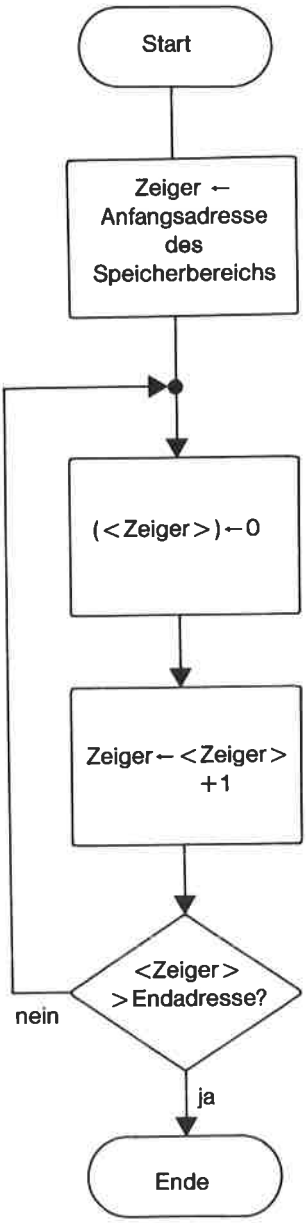


Bild 13.7. *Flußdiagramm: Überschreiben eines Speicherbereichs*

```
LD      (BC),A      ; Speicher mit Null fuellen
INC     BC          ; Zeiger beziehungsweise
                    ; Zaehler erhoehen
LD      HL,4FFFH    ; Ende des Speicherbereichs
SBC     HL,BC       ; auf Ende der Schleife testen
JP      NC,FUELL    ; zum Schleifenanfang springen,
                    ; falls Endwert noch nicht
                    ; ueberschritten
```

Die annehmenden Zählschleifen entsprechen übrigens den FOR-Schleifen in den höheren Programmiersprachen FORTRAN und BASIC.

Übungen

1. Berechne die Summe der ganzen Zahlen von 4 bis 17.

2. Berechne die Summe der ungeraden Zahlen zwischen 0 und 30.

3. Fülle jede zweite Speicherzelle des Bereichs 51A6H-5242H mit dem Wert FFH, wobei die erste gefüllte Speicherzelle die Adresse 51A7H besitzen soll.

4. Für mathematisch Vorgebildete! Berechne folgende Summe:

$$\sum_{i=1}^{n} \quad \sum_{j=1}^{i}$$

5. Realisiere das erste Beispiel dieses Unterkapitels mit Hilfe einer Zählgröße, die im Speicher untergebracht ist.

13.3 Selbstgesteuerte abweisende Zählschleifen

Eine selbstgesteuerte abweisende Zählschleife funktioniert ähnlich wie eine selbstgesteuerte annehmende Zählschleife, jedoch wird zu Anfang geprüft, ob die Schleife überhaupt ausgeführt werden soll. Eine absteigende selbstgesteuerte abweisende Zählschleife besitzt folgende Formalisierung (das Zeichen >= steht für »größer oder gleich«):

wenn Startwert >= Endwert
dann **wiederhole**
Schleifenkörper
mit Schleifenzähler **von** Startwert **bis** Endwert **in Schritten von** Schrittweite

Die Schrittweite ist dabei negativ. Das Programmstück zum Testen des Endkriteriums steht normalerweise am Anfang der Schleife, also vor dem Schleifenkörper. Es ist aber auch möglich, die Schleife als annehmende Schleife zu modellieren und als erstes auf das Ende dieser Schleife zu springen; der Startwert muß dann eventuell vorher um den Betrag der Schrittweite erhöht werden (nämlich wenn wir auf das Programmstück springen, das den neuen Wert des Zählers berechnet). Wir sehen uns beide Varianten an einem Beispiel an:

Es seien m und n zwei positive gerade Zahlen. Wir wollen nun alle geraden Zahlen aufsummieren, die nicht größer als n und nicht kleiner als m sind. Dabei kann auch der Fall n< m auftreten, in dem über eine leere Menge summiert wird. Wir nehmen an, daß m im B-Register und n im C-Register übergeben wird. Die Summe soll im HL-Register gebildet werden.

Das allgemeine Verfahren lautet siehe Bild 13.8.

Wir setzen zuerst das Endkriterium vor den Schleifenkörper:

```
        LD      HL,0        ; Akkumulator loeschen
        LD      D,H         ; D <- 0
        LD      A,C         ; Startwert in Zaehler
                            ; uebertragen
ADD:    CP      B           ; auf Zaehler < m testen
        JP      C,FERTIG    ; Zaehler < m,
                            ; Schleife terminiert
        LD      E,A         ; Summand ins DE-Register bringen
        ADD     HL,DE       ; Summand zu Akkumulator addieren
        SUB     2           ; Schrittweite zu Zaehler
                            ; addieren
        JP      ADD         ; zum Schleifenanfang springen
FERTIG: NOP                 ; Fortsetzungspunkt nach
                            ; Verlassen der Schleife
```

Alternativ dazu steht die Variante mit Einsprung in die Schleife:

```
        LD      HL,0        ; Akkumulator loeschen
        LD      D,H         ; D <- 0
        LD      A,C         ; Startwert in Zaehler
                            ; uebertragen
        JP      EINSPR      ; in Schleife einspringen
ADD:    LD      E,A         ; Summand ins DE-Register bringen
        ADD     HL,DE       ; Summand zu Akkumulator addieren
        SUB     2           ; Schrittweite zu Zaehler
                            ; addieren
EINSPR: CP      B           ; auf Zaehler < m testen
        JP      NC,ADD      ; Zaehler >= m,
                            ; zum Schleifenanfang springen
```

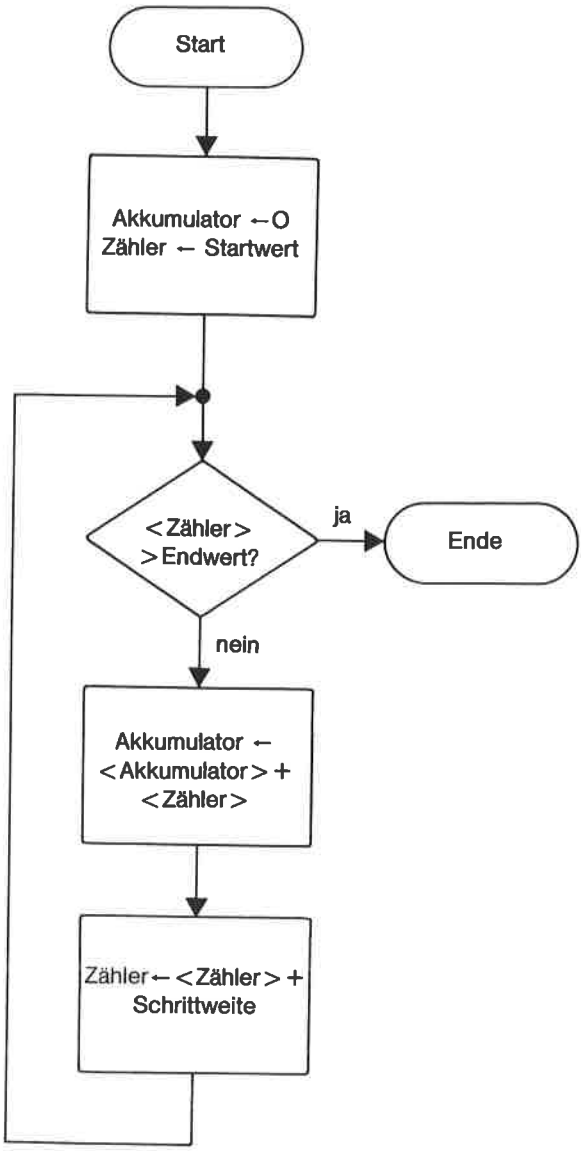


Bild 13.8. *Flußdiagramm: Summe von geraden Zahlen berechnen*

```
FERTIG:   NOP                    ; Fortsetzungspunkt nach
                                 ; Verlassen der Schleife
```

Die aufsteigende selbstgesteuerte abweisende Zählschleife kann formal folgendermaßen dargestellt werden (das Zeichen <= steht dabei für »kleiner oder gleich«):

wenn Startwert <= Endwert
dann **wiederhole**
Schleifenkörper
mit Schleifenzähler **von** Startwert **bis** Endwert **in Schritten von** Schrittweite

Die Schrittweite ist hier positiv. Wir schreiben das vorhergehende Beispiel als aufsteigende selbstgesteuerte abweisende Zählschleife. Im Flußdiagramm, Bild 13.9.

Wir beachten dabei, daß die Schleife genau dann terminiert, wenn die Zählgröße größer als n ist; ist n = 254, so kann die Zählgröße (in diesem Beispiel) in einem 8-Bit-Register nicht größer als n werden, da modulo 256 reduziert wird. Wir müssen deshalb zusätzlich prüfen, ob beim Inkrementieren der Zählgröße ein Übertrag eingetreten ist:

```
          LD      HL,0       ; Akkumulator loeschen
          LD      D,H        ; D <- 0
          LD      A,B        ; Startwert in Zaehler
                             ; uebertragen
ADD:      CP      C          ; auf Zaehler > n testen
          JP      C,KOERP    ; Zaehler < n, Schleifenkoerper
                             ; wird ausgefuehrt
          JP      NZ,FERTIG  ; Zaehler > n,
                             ; Schleife terminiert
KOERP:    LD      E,A        ; Summand ins DE-Register bringen
          ADD     HL,DE      ; Summand zu Akkumulator addieren
          ADD     A,2        ; Schrittweite zu Zaehler
                             ; addieren
          JP      NC,ADD     ; Zaehler < 256,
                             ; Schleife fortsetzen
FERTIG:   NOP                ; Fortsetzungspunkt nach
                             ; Verlassen der Schleife
```

Das Endkriterium dieser Schleife ist ziemlich kompliziert; es besteht aus drei Befehlen, die auf Schleifenanfang und Schleifenende verteilt sind: Eine Prüfung, ob der Schleifenzähler infolge Übertrags logisch größer als n ist, obwohl die Zählgröße kleiner als n ist; eine Prüfung, ob der Schleifenzähler kleiner als n ist; und eine Prüfung, ob der Schleifenzähler gleich n ist.

In der Einsprung-Variante lautet das Programm:

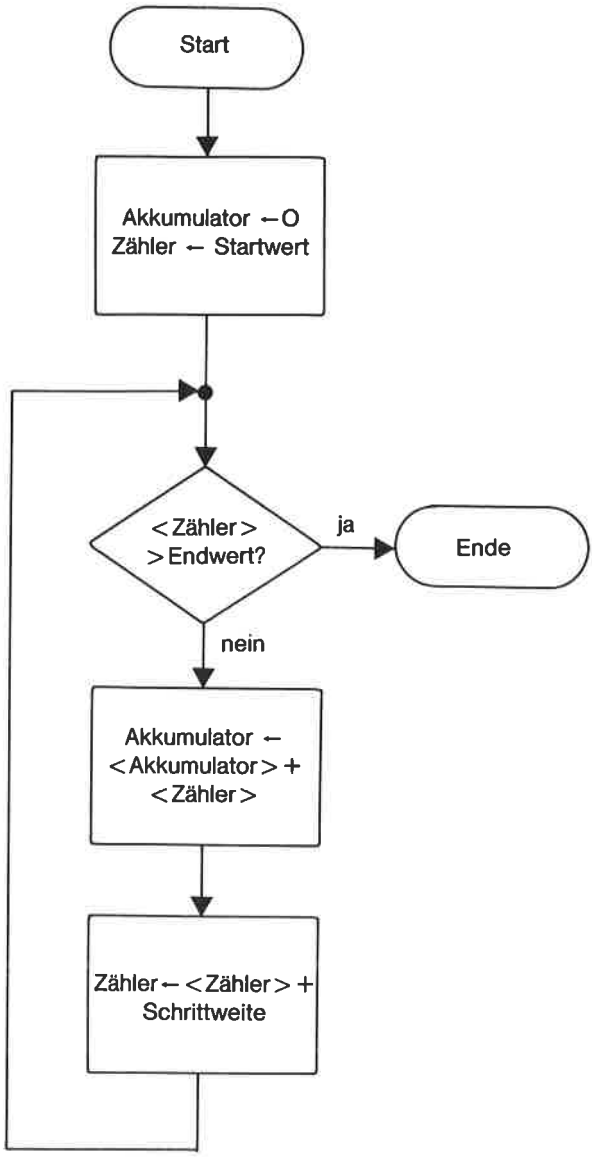


Bild 13.9. *Flußdiagramm: Summation als aufsteigende Zählschleife*

```
        LD      HL,0        ; Akkumulator loeschen
        LD      D,H         ; D <- 0
        LD      A,B         ; Startwert in Zaehler
                            ; uebertragen
        JP      EINSPR      ; in Schleife einspringen
ADD:    LD      E,A         ; Summand ins DE-Register bringen
        ADD     HL,DE       ; Summand zu Akkumulator addieren
        ADD     A,2         ; Schrittweite zu Zaehler
                            ; addieren
        JP      C,FERTIG    ; Zaehler >= 256 > n
EINSPR: CP      C           ; auf Zaehler > n testen
        JP      C,ADD       ; Zaehler < n,
                            ; zum Schleifenanfang springen
        JP      Z,ADD       ; Zaehler = n,
                            ; zum Schleifenanfang springen
FERTIG: NOP                 ; Fortsetzungspunkt nach
                            ; Verlassen der Schleife
```

Die abweisenden Zählschleifen entsprechen den FOR-Schleifen in den höheren Programmiersprachen PASCAL und ALGOL.

Übungen

1. Schreibe ein Programm, das die Summe aller ganzen Zahlen zwischen m und n berechnet.

2. Schreibe ein Programm, das die Summe aller durch drei teilbaren positiven Zahlen berechnet, die nicht größer als eine vorgegebene Zahl n sind.

13.4 Annehmende Schleifen mit allgemeiner Bedingung

Im letzten Beispiel war das Terminierungskriterium relativ kompliziert gestaltet, was die Struktur der Zählschleife schon weitgehend verschleierte. In der Tat ist eine Zählschleife auf Assemblerebene nur ein Spezialfall einer Schleife mit allgemeiner Bedingung. Die Bedingung gibt an, wann die Schleife terminieren soll. Wie schon bei den Zählschleifen gibt es auch hier annehmende Schleifen, deren Schleifenkörper stets mindestens einmal durchlaufen wird, und abweisende Schleifen, bei denen das Terminierungskriterium angibt, ob die Schleife überhaupt betreten wird. Wir behandeln zuerst ein Beispiel für eine annehmende Schleife mit allgemeiner Bedingung (das Terminierungskriterium steht am Ende der Schleife):

Die sogenannten Fibonacci-Zahlen bilden eine unendliche Folge natürlicher Zahlen f_i mit folgendem Bildungsgesetz:

$f_0 = 1$,
$f_1 = 1$,
$f_{i+1} = f_i + f_{i-1}$ für $i > 0$.

Wir wollen nun die Folge der Fibonacci-Zahlen soweit berechnen, bis wir auf eine Zahl größer als 1000 treffen. Die Zahlen f_i beziehungsweise f_{i-1} wollen wir im DE-Register beziehungsweise BC-Register aufbewahren, das neue Folgenglied berechnen wir im HL-Register. Das Endergebnis wird im DE-Register stehen. Zusätzlich zur Berechnung der gewünschten Fibonacci-Zahl f_i wollen wir auch deren Index i bestimmen und im A-Register abliefern. Unser Algorithmus sieht formal so aus:

```
A <- 1                                  ; Index i
BC <- 1                                 ; f(i-1) = f0 = 1
DE <- 1                                 ; fi = f1 = 1
wiederhole
        A <- <A> + 1                    ; Index der nächsten
                                        ; zu berechnenden Zahl
        HL <- <BC> + <DE>               ; f(i+1) = fi + f(i-1)
        BC <- <DE>                      ; neues f(i-1) = altes fi
        DE <- <HL>                      ; neues fi = altes f(i+1)
bis <DE> > 1000                         ; fi > 1000
```

Im Flußdiagramm stellt sich das folgendermaßen dar, siehe Bild 13.10.

Das zugehörige Programm lautet:

```
        LD      BC,1        ; f(i-1) = f0 = 1
        LD      D,B         ; D <- 0
        LD      E,C         ; E <- 1, fi = f1 = 1
        LD      A,C         ; A <- 1, Index i
FIBO:   INC     A           ; Index der naechsten
                            ; zu berechnenden Zahl
        LD      H,B         ; direktes Addieren von
        LD      L,C         ; BC und DE geht nicht
        ADD     HL,DE       ; f(i+1) = fi + f(i-1)
        LD      B,D         ; BC <- <DE>
        LD      C,E         ; neues f(i-1) = altes fi
        LD      D,H         ; DE <- <HL>
        LD      E,L         ; neues fi = altes f(i+1)
        LD      HL,1000     ; Schranke fuer fi
        OR      A           ; ein Trick: CY <- 0
        SBC     HL,DE       ; auf fi > 1000 testen
        JP      NC,FIBO     ; fi <= 1000,
                            ; zum Schleifenanfang springen
```

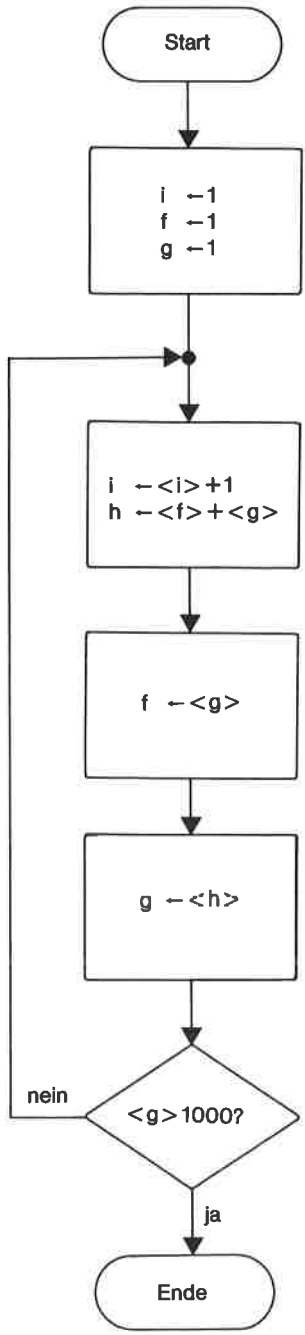


Bild 13.10. *Flußdiagramm: Berechnung von Fibonacci-Zahlen*

Die annehmenden Schleifen mit allgemeiner Bedingung entsprechen den REPEAT-Schleifen der höheren Programmiersprache PASCAL.

Übungen

1. Schreibe ein Programm, das einen bestimmten Speicherbereich aufwärts durchsucht, bis ein vom Leerzeichen verschiedenes Zeichen auftaucht (das erste Zeichen des Speicherbereichs soll dabei nicht angeschaut werden).

2. Berechne zu einer positiven ganzen Zahl x den Exponenten i, für den $3^i <= x < 3^{i+1}$ gilt.

13.5 Abweisende Schleifen mit allgemeiner Bedingung

Wie wir bei den Zählschleifen gesehen haben, kann eine abweisende Schleife durch Einsprung in eine annehmende Schleife oder durch Vorziehen des Terminierungskriteriums vor den Schleifenkörper realisiert werden. Wir betrachten folgendes Beispiel:

Wir wollen den Inhalt des HL-Registers solange zyklisch rechts-rotieren, bis Bit 0 des HL-Registers (also Bit 0 des L-Registers) gesetzt ist. Dabei soll aber höchstens n mal rotiert werden (dies ist sinnvoll, falls man weiß, daß der Inhalt des HL-Registers kleiner als 2^{n+1} ist; insbesondere gilt dies natürlich für n=15). Die Anzahl der Verschiebungen wollen wir im E-Register festhalten. Die Zahl n erwarten wir im A-Register.

Im Flußdiagramm stellt sich das Verfahren folgendermaßen dar, siehe Bild 13.11.

Wir setzen in unserem Programm die Terminierungsbedingung an den Anfang der Schleife:

```
        LD      E,0         ; Zaehler initialisieren
VERSCH: BIT     0,L         ; Bit 0 des HL-Registers testen
        JP      NZ,FERTIG   ; Bit 0 gesetzt,
                            ; Schleife terminiert
        CP      E           ; Zahl der Verschiebungen pruefen
        JP      Z,FERTIG    ; n mal rotiert,
                            ; Schleife terminiert
        SRL     H           ; wir benutzen die Tatsache, dass
        RR      L           ; Bit 7 des H-Registers den alten
                            ; Wert von Bit 0 des L-Registers
                            ; erhalten soll; dieser war Null
        INC     E           ; Anzahl der Verschiebungen
                            ; wurde um 1 erhoeht
        JP      VERSCH      ; zum Schleifenanfang springen
FERTIG: NOP                 ; Fortsetzungspunkt nach
                            ; Verlassen der Schleife
```

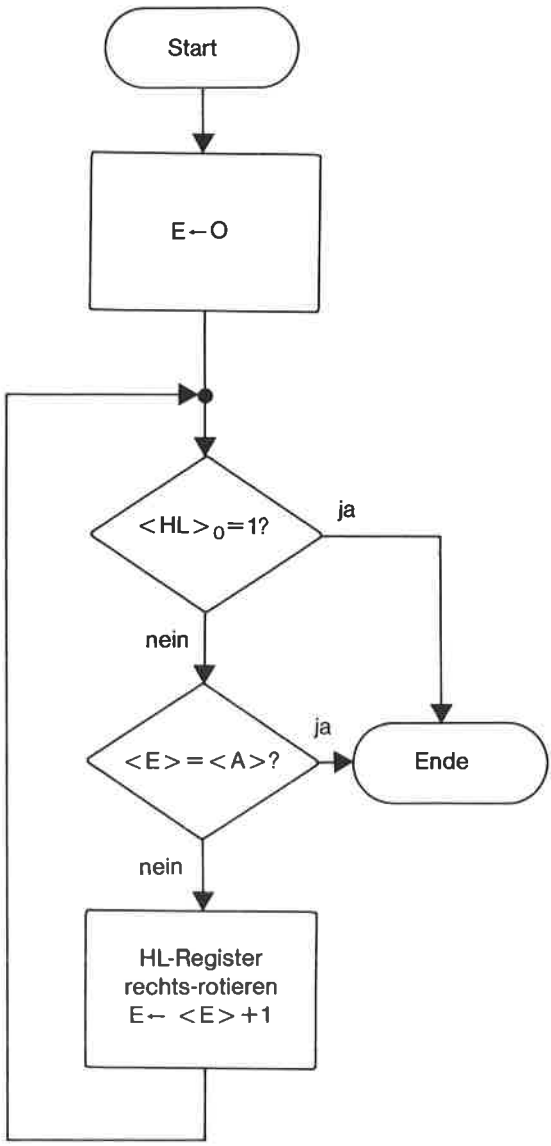


Bild 13.11. *Flußdiagramm: Normalisieren des HL-Registers*

Ob Bit 0 des HL-Registers nun gesetzt ist (oder ob dagegen die Schleife nach n Schritten terminierte, ohne daß Bit 0 des HL-Registers gesetzt war), erkennen wir nach Beendigung der Schleife am Zustand des Null-Flags, das genau dann gelöscht ist, wenn Bit 0 des HL-Registers gesetzt ist.

Um das Beispiel etwas interessanter zu gestalten, bringen wir noch eine Variante mit Ein-

sprung in die Schleife, wobei Gebrauch vom DJNZ-Befehl gemacht wird. Zunächst das Flußdiagramm:

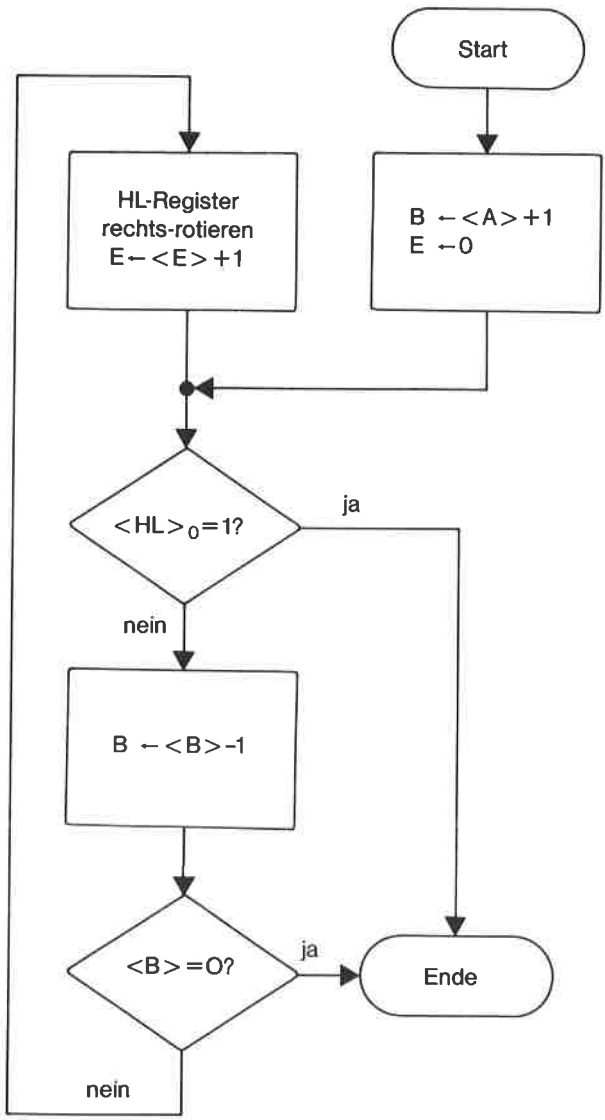


Bild 13.12. *Flußdiagramm: Normalisieren mit dem DJNZ-Befehl*

Hier das zugehörige Programm:

```
        LD      B,A     ; Maximalzahl der Verschiebungen
        INC     B       ; Anpassung wegen DJNZ-Befehl
        LD      E,0     ; Zaehler initialisieren
        JP      EINSPR  ; in Schleife einspringen
VERSCH: SRL     H       ; wir benutzen die Tatsache, dass
        RR      L       ; Bit 7 des H-Registers den alten
                        ; Wert von Bit 0 des L-Registers
                        ; erhalten soll; dieser war Null
        INC     E       ; Anzahl der Verschiebungen
                        ; wurde um 1 erhoeht
EINSPR: BIT     0,L     ; Bit 0 des HL-Registers testen
        JP      NZ,FERTIG ; Bit 0 gesetzt,
                        ; Schleife terminiert
        DJNZ    VERSCH  ; zum Schleifenanfang springen,
                        ; falls noch nicht Maximalzahl
                        ; von Verschiebungen
FERTIG: NOP             ; Fortsetzungspunkt nach
                        ; Verlassen der Schleife
```

Studieren Sie die beiden Programme, bis Sie sie in allen Einzelheiten genau verstanden haben!

Die abweisenden Schleifen mit allgemeiner Bedingung entsprechen den WHILE-Schleifen in der höheren Programmiersprache PASCAL.

Übungen

1. Sehen Sie sich das letzte Programm dieses Unterkapitels nochmals an! Was können Sie am Zustand des Null-Flags erkennen?

2. Verschiebe den Inhalt des A-Registers so lange nach links, bis er größer als der Inhalt des C-Registers geworden ist.

3. Durchsuche einen Speicherbereich abwärts bis zum Zeichen '*'.

13.6 Endlose Schleifen

Eine Schleife mit allgemeiner Bedingung wird verlassen, wenn das Terminierungskriterium erfüllt ist. Nun ist es auch möglich, ein prinzipiell unerfüllbares Kriterium vorzugeben; es entsteht eine Schleife, die niemals terminiert, eine endlose Schleife (das Kriterium kann man dann auch weglassen).

Endlose Schleifen werden benötigt, wenn ein Prozeß immer wieder ohne Einschränkungen ablaufen soll. Nehmen wir zum Beispiel ein Betriebssystem, das nach der Initialisierung in ununterbrochener Folge die Benutzereingaben nach Kommandos absucht und diese ausführt, bis der Rechner ausgeschaltet wird. Oder eine elektronische Uhr. Oder einen vom Computer gesteuerten Regelmechanismus für eine Heizanlage.

Während der Schleifenkörper einer endlosen Schleife beliebig kompliziert sein kann, besteht die Schleifensteuerung nur aus einem an den Schleifenkörper anschließenden Sprung auf den Anfang des Schleifenkörpers:

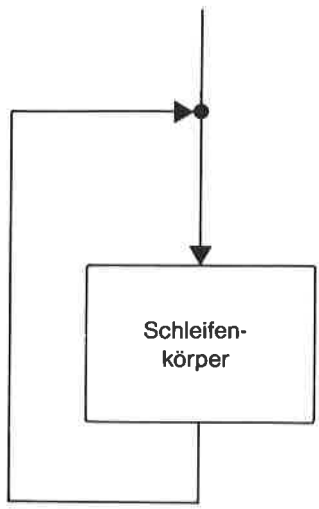


Bild 13.13. *Flußdiagramm: Endlose Schleife*

Das Programm stellen wir uns etwa so vor:

```
ENDLOS:                          ; Schleifenkoerper, zum Beispiel:
                                 ; Lies von der Tastatur
                                 ; fortlaufend Zeichen ein und
                                 ; verarbeite die dadurch
                                 ; dargestellten Kommandos

        JP          ENDLOS       ; Schleifensteuerung:
                                 ; Sprung zum Schleifenanfang
```

13.7 Abbruch von Schleifen

Wir haben uns bisher auf Schleifen beschränkt, die nur dadurch verlassen werden konnten, daß vor beziehungsweise nach Abarbeitung des Schleifenkörpers ein Terminierungskriterium ausgewertet wurde. Vielfach kommt es jedoch beim Abarbeiten des Schleifenkörpers zu Ausnahmesituationen, die das sofortige Verlassen der Schleife - einen Abbruch - wünschenswert erscheinen lassen. Wir realisieren dies durch einen entsprechenden Sprung aus dem Schleifenkörper heraus hinter das Schleifenende.

Wir wollen beispielsweise mit Hilfe einer abweisenden Schleife einen Speicherbereich fortlaufend durchsuchen, bis wir ein bestimmtes Zeichen gefunden haben, oder dabei - ohne das Zeichen zu finden - das Ende des Speicherbereichs überschritten haben. Die Anfangsadresse des Speicherbereichs stehe im HL-Register, die Endadresse im DE-Register. Das Zeichen bekommen wir im A-Register geliefert. Unser Terminierungskriterium (nächste Adresse > Endadresse) setzen wir an den Anfang der Schleife. Ein Abbruch erfolgt, falls wir das gewünschte Zeichen im Speicher gefunden haben. Das allgemeine Verfahren lautet:

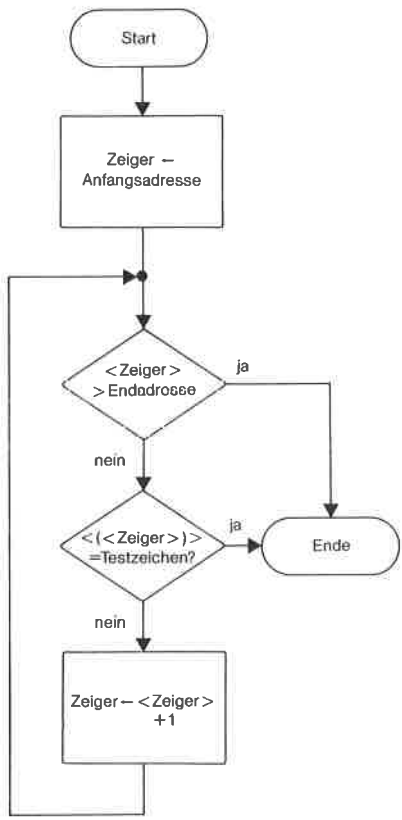


Bild 13.14. *Flußdiagramm: Speicherbereich absuchen*

Als Programm erhalten wir:

```
SUCHE:   LD      B,H        ; HL-Register
         LD      C,L        ; sichern
         OR      A          ; CY <-- 0
         SBC     HL,DE      ; naechste Adresse gegen
                            ; Endadresse testen
         JP      C,VERGL    ; naechste Adresse < Endadresse,
                            ; Schleife fortsetzen
         JP      NZ,FERTIG  ; naechste Adresse > Endadresse,
                            ; Schleife terminiert
VERGL:   LD      H,B        ; HL-Register
         LD      L,C        ; wiederherstellen
         CP      (HL)       ; Zeichen im Speicher mit
                            ; Testzeichen vergleichen
         JP      Z,FERTIG   ; Abbruch der Schleife, da
                            ; gewuenschtes Zeichen gefunden
         INC     HL         ; auf naechstes Zeichen zeigen
         JP      SUCHE      ; zum Schleifenanfang springen
FERTIG:  NOP                ; Fortsetzungspunkt nach
                            ; Verlassen der Schleife
```

Am Zustand des Null-Flags können wir übrigens erkennen, ob wir das gewünschte Zeichen gefunden haben (wie?).

Das Fehlen eines expliziten Konstrukts für den Abbruch von Schleifen ist einer der großen Mängel vieler höherer Programmiersprachen (ALGOL, PASCAL, BASIC). Wer sich für Abbruch-Konstrukte in höheren Programmiersprachen interessiert, kann sich als Beispiel das EXIT-Konstrukt der Sprache muSIMP (einer LISP-Variante) ansehen.

Wir haben nun einige Formen von Schleifen kennengelernt, die eine ziemlich klare Ablaufstruktur besitzen. In der Regel kommt man damit auch aus. Allerdings ist es unter Umständen möglich, durch Kombination der vorgestellten Schleifentypen optimalere Programme zu erhalten. Darauf soll an dieser Stelle aber nicht weiter eingegangen werden.

Übungen

1. Im letzten Beispielprogramm mußten wir in das Überprüfen der Bedingung »nächste Adresse > Endadresse« zwei Sprungbefehle investieren. Überlegen Sie sich Modifikationen des Programms, die den zweiten Sprungbefehl überflüssig machen.

2. Multipliziere zwei vorzeichenlose ganze 8-Bit-Zahlen; brich dabei ab, falls das Ergebnis nicht als 8-Bit-Größe darstellbar ist.

3. Berechne zu einer positiven ganzen Zahl i die Potenz 2^i; brich ab, wenn das Ergebnis nicht mit 16 Bits darstellbar ist.

14
Felder

Ein (eindimensionales) Feld ist eine fortlaufend indizierte Menge von Datenwerten gleichen Typs (zum Beispiel vom Typ Byte, Wort, Bit, Nibble); ein Feld von Zahlen können wir uns als Vektor vorstellen. Die Indizes sind ganze Zahlen. Es gibt einen kleinsten Index i_u und einen größten Index i_o. Jedem Index im Bereich i_u bis i_o ist ein Feldelement zugeordnet.

14.1 Implementierung von Feldern

Wir nehmen stets an (was in der Praxis fast immer stimmt), daß alle Feldelemente lückenlos im Speicher untergebracht sind. Wir beginnen zunächst mit Feldern, deren Elemente jeweils ein oder mehrere Bytes belegen (eine glatt durch 8 teilbare Anzahl von Bits). Ist »L« die Länge eines einzelnen Feldelements, so benötigt das ganze Feld $(i_o - i_u + 1) * $ »L« Bytes Speicherplatz. Für ein uninitialisiertes Feld stellen wir diesen durch die DEFS-Pseudo-Operation bereit, zum Beispiel für ein Feld von Worten mit dem Indexbereich 0 bis 4:

```
IU        EQU     0                   ; kleinster Index
IO        EQU     4                   ; groesster Index
LAENGE    EQU     2                   ; Laenge eines Elements
WFELD:    DEFS    (IO-IU+1)*LAENGE    ; uninitialisiertes Feld
                                      ; von Worten reservieren
```

Wollen wir ein initialisiertes Feld schaffen, so schreiben wir die Elemente mittels der Pseudo-Operationen DEFB und DEFW hintereinander auf; in Fortsetzung des Beispiels also:

```
WFELD:    DEFW    19738       ; erstes Feldelement
          DEFW    7895        ; zweites Feldelement
```

```
        DEFW    -3396       ; drittes Feldelement
        DEFW    12987       ; viertes Feldelement
        DEFW    24745       ; fuenftes Feldelement
```

Sind die Feldelemente weder Bytes noch Worte, so setzt sich die Vereinbarung eines Elements aus mehreren Pseudo-Operationen zusammen, also zum Beispiel für ein 4elementiges Feld, dessen Elemente je 3 Bytes belegen:

```
FELD3:  DEFB    22H         ; erstes Feldelement
        DEFB    45H
        DEFB    8AH
        DEFB    0AH         ; zweites Feldelement
        DEFB    19H
        DEFB    0C7H
        DEFB    31H         ; drittes Feldelement
        DEFB    86H
        DEFB    73H
        DEFB    00H         ; viertes Feldelement
        DEFB    1FH
        DEFB    27H
```

Für Zeichen und Zeichenketten verwenden wir die Pseudo-Operationen DEFB beziehungsweise DEFM.

Belegen die Elemente eines Felds je »L« Bits und ist »L« nicht glatt durch 8 teilbar, so gibt es zwei Möglichkeiten: Entweder verschenkt man pro Element einige Bits und teilt jedem Element so viele Bytes zu wie zur Aufnahme von »L« Bits benötigt werden; die Darstellung des Felds geschieht dann wie oben beschrieben. Oder man faßt den Speicher als eine Folge von Bits auf und teilt jedem Element genau »L« Bits Speicherplatz zu; die Grenzen der Feldelemente fallen dann nicht immer mit den Grenzen von Bytes zusammen. Wir betrachten in Zukunft nur diesen zweiten Fall. Als gesamten Speicherplatz des Felds wählt man die kleinste Anzahl von Bytes, die zur Aufnahme von $(i_o - i_u + 1) * $ »L« Bits notwendig sind. Beispielsweise vereinbart man ein uninitialisiertes Feld von 17 Nibbles mittels

```
NFELD:  DEFS    9           ; Feld mit 17 Nibbles reservieren
```

und ein Bit-Feld mit 116 Elementen mittels

```
BFELD:  DEFS    15          ; Feld mit 116 Bits reservieren
```

Initialisierte Felder dieser Art werden wieder mit DEFB und DEFW vereinbart, also zum Beispiel für den Nibble-Vektor (4,7,9,A,F):

```
NVEK:   DEFB    74H         ; initialisiertes Feld
        DEFB    0A9H        ; mit 5 Nibbles
        DEFB    0FH         ; vereinbaren
```

Achte darauf, wie die einzelnen Elemente als höherwertige und niederwertige Anteile von Bytes plaziert werden!

Bisher haben wir stets über eindimensionale Felder gesprochen. Es gibt jedoch auch mehrdimensionale Felder, zum Beispiel stellt man eine Matrix durch ein zweidimensionales Feld dar. Mehrdimensionale Felder werden durch ein Index-Tupel indiziert, zum Beispiel durch das Paar (i,j) oder das Tripel (i,j,k). Ist d die Dimension des Felds, so enthält jedes Index-Tupel die d-Indizes $i_1, i_2, ..., i_d$, die den Bereichen $i_{u_1} - i_{o_1}, i_{u_2} - i_{o_2}, ..., i_{u_d} - i_{o_d}$ entnommen sind. Wir beschränken uns auf den für die Praxis relevanten Spezialfall d=2.

Stellen wir uns ein zweidimensionales Feld als Darstellung einer Matrix von Elementen vor, so gibt der eine Index eine Zeile, der andere eine Spalte der Matrix an. Damit gibt es prinzipiell zwei Möglichkeiten, wie das Feld organisiert sein kann: Entweder stehen alle Elemente einer Zeile direkt hintereinander (zeilenorientiertes Feld) oder aber alle Elemente einer Spalte (spaltenorientiertes Feld). Wir können das zweidimensionale Feld im ersten Fall als eindimensionales Feld von Zeilen, im zweiten Fall als eindimensionales Feld von Spalten interpretieren, wobei Zeilen und Spalten wiederum eindimensionale Felder von Elementen darstellen. Wir geben hierzu ein kleines Beispiel. Unsere Matrix soll folgendermaßen aussehen:

$$\begin{pmatrix} 12 & -17 & 0 \\ 0 & 33 & 8 \end{pmatrix}$$

Wir vereinbaren ein zeilenorientiertes Byte-Feld durch

```
ZFELD:  DEFB    12          ; erste Zeile
        DEFB    -17
        DEFB    0
        DEFB    0           ; zweite Zeile
        DEFB    33
        DEFB    8
```

und ein spaltenorientiertes Byte-Feld durch

```
SFELD:  DEFB    12          ; erste Spalte
        DEFB    0
        DEFB    -17         ; zweite Spalte
        DEFB    33
        DEFB    0           ; dritte Spalte
        DEFB    8
```

Für andere Basistypen (Wort, Bit, Nibble) oder größere Dimension d funktioniert dies alles ganz analog.

Der gesamte Platzbedarf eines d-dimensionalen Felds ist »L«

$$* \prod_{j=1}^{d} (i_{o_j} - i_{u_j} + 1)$$

Bytes (beziehungsweise Bits), wenn »L« die Länge eines Feldelements in Bytes (beziehungsweise Bits) ist. Für d=2 vereinfacht sich dies zu »L« * (i_{o_1}-i_{u_1}+1) * (i_{o_2}-i_{u_2}+1).

Die Dimension eines Felds, die Unter- und Obergrenzen der Indexbereiche, und der Typ der Elemente gehören eigentlich untrennbar von den Elementen selbst (beziehungsweise den Speicherplätzen dafür) zur Definition eines Felds. Diese Informationen sind meist im Programm versteckt: als Konstanten, die im Algorithmus auftauchen, wenn nur ein Feld mit unveränderlichen Abmessungen bearbeitet wird; als Registerinhalte, die zur Laufzeit des Programms berechnet werden, wenn sich die Indexgrenzen während des Programmlaufs verändern. Wenn wir Programme schreiben wollen, die auf Feldern beliebiger Größe arbeiten sollen, so fassen wir die Strukturinformationen zu einem sogenannten Deskriptor zusammen, den wir vor die Elemente plazieren. Im Falle unserer Matrix ZFELD würde dies beispielsweise lauten:

```
ZFELD:
TYP:    DEFB    1       ;jedes Element belegt 1 Byte
DIM:    DEFB    2       ; Dimension d=2
IU1:    DEFB    1       ; Zeilenindex laeuft
IO1:    DEFB    2       ; von 1 bis 2 .
IU2:    DEFB    1       ; Spaltenindex laeuft
IO2:    DEFB    3       ; von 1 bis 3
ELEM:   DEFB    12      ; erste Zeile
        DEFB    -17
        DEFB    0
        DEFB    0       ; zweite Zeile
        DEFB    33
        DEFB    8
```

Um die Deskriptoreinträge richtig zu interpretieren, müssen wir natürlich die Struktur und die Bedeutung des Deskriptors kennen (zum Beispiel müssen wir wissen, daß als erstes die Länge eines Elements, gemessen in Bytes, angegeben wird); außerdem müssen wir für obiges Beispiel die Vereinbarung treffen, daß die Matrix zeilenorientiert ist.

Übungen

1. Vereinbare folgende uninitialisierten eindimensionalen Felder:

kleinster Index	größter Index	Basistyp/Länge
1	10	Wort
0	15	Byte
−5	5	4 Bytes
1	25	Bit
0	12	Nibble
0	31	2 Bits

2. Vereinbare folgende initialisierten Vektoren:

Typ	Vektor
Byte	(8, −13, 17, 99, −121, 44)
Wort	(12380, 16421, 246, −13131)
Bit	(0,0,0,1,1,0,1,1,1,1,0,1,0)
Nibble	(3, A, 7, 0, B, F, 1, 4, D)

3. Vereinbare folgende Matrizen einmal als zeilenorientiertes und einmal als spaltenorientiertes Feld:

Typ	Matrix
Wort	$\begin{pmatrix} 4711 & -21979 \\ 12391 & -1731 \end{pmatrix}$
Bit	$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$

Wieviel Speicherplatz benötigen die Matrizen?
Gib für die spaltenorientierte Wort-Matrix eine Vereinbarung mit Deskriptor an!

14.2 Adressierung einzelner Feldelemente

Wir behandeln zunächst wieder eindimensionale Felder, deren Elemente je ein ganzzahliges Vielfaches von 8 Bits belegen. Um ein einzelnes Element mit Index i zu bearbeiten, benötigen wir die Anfangsadresse des Elements (es geht prinzipiell auch mit der Endadresse; wir wollen uns aber hier auf Überlegungen zur Anfangsadresse beschränken). Die Speicherabbildungsfunktion a liefert zu jedem Index i die Anfangsadresse a(i) des i-ten Feldelements. Aus der im vorhergehenden Unterkapitel vereinbarten Speichertechnik folgt a(i) = a(0) + i * »L«. Meist kennt man aber nicht die Größe a(0), sondern die Anfangsadresse $a_u = a(i_u)$ des Felds. Damit ergibt sich a(i) = a_u + (i−i_u) * »L«.

Vom Index i zur Adresse a(i) gelangen wir nun in drei Schritten:

1. Berechnung des relativen Index i−i_u (entfällt im Spezialfall i_u=0).
2. Berechnung der relativen Adresse (i−i_u) * »L« (entfällt im Spezialfall »L«=1, also bei Byte-Feldern).
3. Berechnung der absoluten Adresse a_u + (i−i_u) * »L«.

Haben wir die Adresse a(i) schließlich berechnet, so kann das Feldelement mit den bekannten Methoden der indirekten Adressierung bearbeitet werden.

Als Zeiger auf ein einzelnes Feldelement eignet sich besonders das HL-Register, bedingt auch das BC- oder DE-Register. Wir wählen im folgenden stets das HL-Register zur Datenadressierung.

Wir gehen davon aus, daß der Index i als vorzeichenlose ganze Zahl im A-Register steht; Anpassungen der im weiteren folgenden Programme an Indizes in 2-Komplement-Darstellung und/oder Indizes in einem Doppelregister sind leicht zu bewerkstelligen. Unsere drei Schritte lauten dann schematisch:

A <– <A> – i_u
HL <– <A> * »L«
HL <– a_u + <HL>

Sind i_u oder »L« als Konstanten gegeben, so kann man sie direkt in den Algorithmus einbringen; ansonsten taucht an ihrer Stelle ein geeignetes Register auf. Für den letzten Schritt muß ein weiteres Doppelregister benutzt werden, das die Adresse a_u aufnimmt. Ist »L« eine Konstante, so führen wir den zweiten Schritt als gestreckte Multiplikation (siehe Kapitel »Sequenzen«) aus; ansonsten verwenden wir eine Multiplikationsschleife (siehe Kapitel »Schleifen«).

Wir beginnen mit einem Byte-Feld (»L«=1), wobei wir i_u und a_u als Konstanten vorgeben:

```
IU      EQU     12          ; kleinster Index
AU      EQU     4200H       ; Anfangsadresse des Felds
        SUB     IU          ; relativen Index berechnen
        LD      L,A         ; relativer Index = relative
        LD      H,0         ; Adresse wegen »L«=1
        LD      DE,AU       ; Anfangsadresse des Felds
        ADD     HL,DE       ; Adresse des Feldelements
```

Nun dasselbe für ein Wort-Feld:

```
IU      EQU     5           ; kleinster Index
AU      EQU     5E00H       ; Anfangsadresse des Felds
        SUB     IU          ; relativen Index berechnen
        LD      L,A         ; relativen Index ins
        LD      H,0         ; HL-Register bringen
        ADD     HL,HL       ; relative Adresse berechnen
        LD      DE,AU       ; Anfangsadresse des Felds
        ADD     HL,DE       ; Adresse des Feldelements
```

Nun noch eine Routine für die Indizierung, wenn die Kenngrößen des Felds in Registern übergeben werden. Wir erwarten dabei den Index i im A-Register, den kleinsten Index i_u im C-Register, die Länge »L« im E-Register und die Anfangsadresse a_u des Felds im HL-Register. a_u müs-

sen wir temporär im Speicher ablegen, da wir alle Doppelregister für die Multiplikation benötigen. Die Multiplikationsroutine ist dem Kapitel »Schleifen« entnommen (mit kleinen Änderungen).

```
; Berechnung der Anfangsadresse eines Feldelements
; A = Index i
; C = kleinster Index i_u
; E = Laenge »L« eines Feldelements
; HL = Anfangsadresse des Felds

ADRESS:   LD      (BASIS),HL  ; Registerinhalt temporaer
                              ; sichern

; relativen Index berechnen

          SUB     C           ; relativen Index berechnen

; Multiplikation (i–i_u) *l durchfuehren, das heisst
; relative Adresse des Feldelements berechnen

          LD      HL,0        ; Akkumulator loeschen
          LD      D,H         ; Multiplikand zu 16-Bit-
                              ; Groesse erweitern
          LD      B,8         ; Schleifenzaehler mit Laenge
                              ; des Multiplikators (in Bits)
                              ; besetzen
MULTI:    ADD     HL,HL       ; Akkumulator verdoppeln
          RLCA                ; hoechstes Bit des
                              ; Multiplikators ins Uebertrag
                              ; Flag bringen, gleichzeitig
                              ; Multiplikator links-rotieren
          JP      NC,NULL     ; keine Addition erforderlich,
                              ; wenn anstehendes Bit des
                              ; Multiplikators 0 ist
          ADD     HL,DE       ; Multiplikand zu Akkumulator
                              ; addieren
NULL:     DJNZ    MULTI       ; naechstes Bit des
                              ; Multiplikators verarbeiten

; Anfangsadresse des Feldelements berechnen

          LD      DE,(BASIS)  ; Anfangsadresse des Felds
          ADD     HL,DE       ; Adresse des Feldelements
                              ; berechnen
```

```
; Daten-Bereich

BASIS:    DEFS      2          ; Hilfs-Speicherplatz
```

Nehmen Sie sich bei Ihren eigenen Programmen auch die Zeit, sie so sorgfältig zu dokumentieren, es lohnt sich wirklich!

Bisher haben wir stets angenommen, daß der Index i der Bedingung $i_u <= i <= i_o$ genügt. Durch Programmierfehler, fehlerhafte Daten oder unsinnige Benutzereingaben kann es aber durchaus vorkommen, daß i außerhalb des erlaubten Bereichs liegt. Wir sollten also - um uns vor bösen Überraschungen zu schützen - dynamische Kontrollen einbauen, welche die Einhaltung der Indexgrenzen überwachen. Die Prüfung besteht aus zwei Teilen:

1. Prüfe $i >= i_u$ nach (entfällt für i_u=0).
2. Prüfe $i <= i_o$ nach (entfällt für i_o=255 bei 8-Bit-Indizes).

Ersetzt man den SUB-Befehl zu Beginn des eben gezeigten Programms durch folgendes Programmstück, so kann nichts mehr schiefgehen (i_o übergeben wir im D-Register):

```
DYNKON:   CP        D          ; i mit io vergleichen
          JP        C,OK       ; i < io, Index korrekt
          JP        NZ,UEBERL  ; Indexueberlauf i > io
OK:       SUB       C          ; relativen Index berechnen
          JP        C,UNTERL   ; Indexunterlauf i < iu
```

Manchmal wird statt des größten zulässigen Index die Adresse $a_o = a_u + (i_o - i_u + 1) * $»L« angegeben, die das erste Byte hinter dem Speicherbereich des Felds angibt. Einen Indexüberlauf prüfen wir dann nach Berechnung der Adresse des Feldelements durch folgendes Programmstück, wobei ins DE-Register zunächst die Adresse a_o geladen werden muß:

```
          EX        DE,HL      ; Operanden tauschen zwecks
                               ; einfacherem Vergleich
          SCF                  ; CY <- 1, ebenfalls zwecks
                               ; einfacherem Vergleich
          SBC       HL,DE      ; ao > Adresse des
                               ; Feldelements ?
          JP        C,UEBERL   ; Indexueberlauf
          EX        DE,HL      ; Operanden ruecktauschen
```

Diese Vorgehensweise wird auf jeden Fall notwendig, wenn wir statt des Index i bereits den relativen Index $i-i_u$ erhalten, die Untergrenze i_u uns dagegen nicht bekannt ist (bei vorzeichenlosen ganzzahligen Indizes kann dann kein Unterlauf eintreten).

Der neue Befehl **EX** (exchange) tauscht die Inhalte des DE-Registers und des HL-Registers wechselseitig aus. Für Adressierungsvorgänge werden wir ihn häufig verwenden.

Nun zu den mehrdimensionalen Feldern: Ein zweidimensionales Feld mit den Indexgrenzen $i_{u1} - i_{o1}$ und $i_{u2} - i_{o2}$ können wir uns im Speicher durch ein eindimensionales Feld mit den Indexgrenzen $i_u = 0$, $i_o = (i_{o1} - i_{u1} + 1) * (i_{o2} - i_{u2} + 1) - 1$ vorstellen. Ist das Feld zeilenorientiert, und gibt der erste Index die Zeile, der zweite die Spalte an, so entspricht der Adresse $a(i_1,i_2)$ eines Feldelements in der Zeile i_1 und der Spalte i_2 die Adresse a(i) des Feldelements im eindimensionalen Feld mit $i = (i_1 - i_{u1}) * (i_{o2} - i_{u2} + 1) + (i_2 - i_{u2})$. Haben wir erst den korrespondierenden Index des eindimensionalen Feldes bestimmt, so können wir die dafür entwickelten Methoden anwenden. Diese Abbildung eines d-dimensionalen Felds auf ein eindimensionales Feld funktioniert natürlich nicht nur für Byte-Felder, sondern ebenso für beliebige Nibble- und Bit-Felder. Wir betrachten deshalb für den Rest dieses Unterkapitels nur noch eindimensionale Felder (ohne Deskriptor; auch Indexgrenzenüberwachung sparen wir uns aus Platzgründen).

Wir befassen uns nun als nächstes mit Nibble-Feldern. Da jedes Byte zwei Nibble enthält, kann das i-te Element eines Nibble-Felds durch die Adresse a(i) des Bytes, in dem der Nibble liegt, und die Nibble-Adresse n(i) gekennzeichnet werden; der niederwertige Nibble (Bit 0 - 3) besitzt die Nibble-Adresse 0, der höherwertige Nibble (Bit 4 - 7) die Nibble-Adresse 1. Es gilt damit $2 * a(i) + n(i) = 2 * a(0) + n(0) + i = 2 * a_u + n_u + (i - i_u)$, wobei $a_u = a(i_u)$ die Adresse des ersten Feldelements ist und $n_u = n(i_u)$ dessen Nibble-Adresse. Aus diesen Beziehungen errechnen wir nun $a(i) = a_u + (n_u + (i - i_u)) / 2$ und $n(i) = (n_u + (i - i_u)) \bmod 2$; als Ergebnis der Division ist dabei nur der ganzzahlige Anteil (ohne Rest) zu nehmen. Die Berechnung der Adresse a(i) und der Nibble-Adresse n(i) geht nun so vor sich:

1. Berechnung des relativen Index $i - i_u$.
2. Berechnung des korrigierten relativen Index $n_u + i - i_u$.
3. Gleichzeitige Berechnung von $(n_u + (i - i_u)) / 2$ und $(n_u + (i - i_u)) \bmod 2$.
4. Berechnung der Adresse $a_u + (n_u + (i - i_u)) / 2$.

Berechnung von ganzzahligem Anteil und Rest der Division durch 2 realisieren wir durch eine Rechts-Verschiebung. Dabei fällt der ganzzahlige Anteil im verschobenen Register an, der Rest im Übertrag-Flag; den Wert des Flags müssen wir dann irgendwo anders sichern, damit er durch Schritt 4 nicht zerstört wird.

Wir zeigen nun ein Programmstück für die Realisierung der Schritte 1 bis 3 (Schritt 4 dürfte mittlerweile wohl klar sein). Wir nehmen an, daß alle benötigten Größen in Registern stehen, und zwar i im A-Register, i_u im B-Register und n_u im C-Register. Wir wollen am Ende der Routine die Relativadresse $(n_u + (i - i_u)) / 2$ im A-Register und die Nibble-Adresse n(i) im D-Register stehen haben. Wir nehmen an, daß $i - i_u < 256$ gilt (sonst wird die Arbeit ein bißchen umständlicher). Trotzdem kann natürlich bei der Addition $n_u + (i - i_u)$ ein Übertrag anfallen; damit wir beim Rechts-Verschieben das Ergebnis nicht verfälschen, schieben wir das Übertrag-Flag einfach dabei in Bit 7 hinein - wir rotieren den Akkumulator nach rechts! Das ins Übertrag-Flag herausgeschobene Bit 0 bringen wir dann durch eine Links-Rotation des D-Registers auf dessen Bit 0; damit steht der Rest richtig im D-Register. Das Programmstück lautet also:

```
SUB     B       ; relativen Index berechnen
ADD     A,C     ; korrigierten Index berechnen
```

```
        RRA             ; Division durch 2 und
                        ; Restbildung auf dem
                        ; Superregister CY & A
        LD      D,0     ; vorbereiten zur Speicherung
                        ; der Nibble-Adresse
        RL      D       ; Nibble-Adresse abspeichern
```

Das Ganze funktioniert auch noch, wenn i und/oder i_u in 2-Komplement-Darstellung vorliegen, da ja die Differenz $i-i_u$ laut Definition unseres Felds nicht negativ sein kann.

Der Zugriff auf das i-te Element eines Nibble-Felds geschieht nun folgendermaßen: Wollen wir den Nibble lesen, so holen wir mittels indirekter Adressierung das ihn enthaltende Byte mit der Adresse a(i); sodann isolieren wir den Nibble mit den gelernten Techniken daraus, wobei je nach Wert der Nibble-Adresse n(i) der niederwertige oder der höherwertige Nibble isoliert werden muß. Wollen wir dagegen den Nibble beschreiben, so müssen wir darauf achten, daß der andere Nibble des Bytes nicht zerstört wird. Wir holen am besten erst einmal das ganze Byte mit der Adresse a(i) in ein Register, maskieren den zu ersetzenden Nibble aus, maskieren den neuen Wert des Nibbles wieder ein und bringen das Byte in den Speicher zurück. Wir führen eine solche Schreiboperation als Programmstück vor, wobei wir die Adresse a(i) im HL-Register, die Nibble-Adresse n(i) im D-Register und den einzufügenden Nibble im E-Register erwarten:

```
SNIBB:  LD      A,(HL)  ; beide alten Nibbles besorgen
        DEC     D       ; Nibble-Adresse testen
        JP      Z,EINS  ; Springe, wenn n(i)=1
NULL:   AND     0F0H    ; hinteren Nibble ausmaskieren
        OR      E       ; hinteren Nibble einfuegen
        JP      WEITER  ; weiter an gemeinsamer
                        ; Fortsetzungsstelle
EINS:   AND     0FH     ; vorderen Nibble ausmaskieren
        SLA     E       ; neuen
        SLA     E       ; Nibble
        SLA     E       ; nach vorne
        SLA     E       ; bringen
        OR      E       ; vorderen Nibble einfuegen
WEITER: LD      (HL),A  ; beide Nibbles zurueckschreiben
```

Wir kommen nun abschließend auf Bit-Felder (oder Bitketten, wie man auch sagt) zu sprechen. Ein einzelnes Bit mit dem Index i wird durch die Adresse a(i) des Bytes, in dem das Bit enthalten ist, und die Bit-Adresse (Bit-Nummer) b(i) gekennzeichnet. Bezeichnen wir mit $a_u = a(i_u)$ die Adresse und mit $b_u = b(i_u)$ die Bit-Adresse des ersten Feldelements, so gilt die Beziehung $8 * a(i) + b(i) = 8 * a_u + b_u + (i-i_u)$. Daraus erhalten wir dann $a(i) = a_u + (b_u + (i-i_u)) / 8$ und $b(i) = (b_u + (i-i_u)) \bmod 8$, wobei das Ergebnis der Division durch 8 wieder nur den ganzzahligen Anteil bezeichnet. Die Berechnung der Adresse a(i) und der Bit-Adresse b(i) führen wir wieder in 4 Schritten durch:

1. Berechnung des relativen Index $i-i_u$.
2. Berechnung des korrigierten relativen Index $b_u + i-i_u$.
3. Gleichzeitige Berechnung von $(b_u + (i-i_u)) / 8$ und $(b_u + (i-i_u)) \bmod 8$.
4. Berechnung der Adresse $a_u + (b_u + (i-i_u)) / 8$.

Es ist damit alles wie bei der Adressierung von Nibble-Feldern, nur daß eben die Größen b_u und b(i) je drei Bits belegen und daß statt einer Division durch 2 eine Division durch 8 durchgeführt wird. Wir setzen deshalb die Algorithmen für die Schritte 1, 2 und 4 als bekannt voraus und betrachten die Realisierung von Schritt 3.

Eine Division durch 8 können wir interpretieren als dreimalige Ausführung einer Division durch 2. Wir wenden also wieder die Schiebetechnik an. Die drei Bits, die beim Rechts-Schieben des Ausdrucks $b_u + (i-i_u)$ herausfallen, müssen für b(i) gesammelt und stellenwertkorrekt abgelegt werden. Wir schieben sie also nach rechts in das Register, das b(i) aufnehmen soll, und bringen sie durch 3 weitere zirkuläre Links-Rotationen auf den richtigen Platz (nicht verzweifeln!). Nehmen wir also an, daß $b_u + (i-i_u)$ im A-Register geliefert wird, der ganzzahlige Anteil bei der Division im A-Register stehen bleiben soll und der Rest b(i) ins B-Register kommt, so erreichen wir unser Ziel mit folgendem Programmstück:

```
DIVMOD:  LD    B,0     ; B-Register zur Aufnahme
                       ; von b(i) vorbereiten
         SRL   A       ; durch 2 dividieren
         RR    B       ; herausgefallenes Bit sichern
         SRL   A       ; durch 2 dividieren
         RR    B       ; herausgefallenes Bit sichern
         SRL   A       ; durch 2 dividieren
         RR    B       ; herausgefallenes Bit sichern
         RLC   B       ; Bits an
         RLC   B       ; richtigen Platz
         RLC   B       ; bringen
```

Das zuletzt herausgeschobene Bit können wir auch gleich auf Bit-Position 0 stellen, um einen Befehl einzusparen:

```
DIVMOD:  LD    B,0     ; B-Register zur Aufnahme
                       ; von b(i) vorbereiten
         SRL   A       ; durch 2 dividieren
         RR    B       ; herausgefallenes Bit sichern
         SRL   A       ; durch 2 dividieren
         RR    B       ; herausgefallenes Bit sichern
         SRL   A       ; durch 2 dividieren
         RL    B       ; herausgefallenes Bit sichern
         RL    B       ; Bits an richtigen
         RL    B       ; Platz bringen
```

Die Bearbeitung des Feldelements geschieht am besten durch die Techniken, die wir im Unterkapitel »Maskieren« besprochen haben. Mit Hilfe der Größe b(i), die wir genau deswegen ins B-Register gebracht haben, bauen wir uns eine geeignete Maske für die gewünschte Operation (testen, setzen, rücksetzen, invertieren) auf. Für das Testen (sowie für das Setzen und das Invertieren; alle drei Masken sehen gleich aus - vergleiche Unterkapitel 12.4!):

```
        LD      A,10000000B  ; Bit 7 setzen, Rest loeschen
        INC     B            ; wegen DJNZ-Befehl und
                             ; moeglichem Inhalt 0
SETZEN: RLCA                 ; gesetztes Bit wandern lassen,
        DJNZ    SETZEN       ; bis es am richtigen Platz ist
```

Für das Löschen brauchen wir nur eine andere Ausgangsmaske:

```
        LD      A,01111111B  ; Bit 7 loeschen, Rest setzen
        INC     B            ; wegen DJNZ-Befehl und
                             ; moeglichem Inhalt 0
LOESCH: RLCA                 ; geloeschtes Bit wandern lassen,
        DJNZ    LOESCH       ; bis es am richtigen Platz ist
```

Haben wir die Maske endlich im A-Register stehen, so führen wir - mit der Adresse a(i) im HL-Register - die Operation folgendermaßen aus:

Für das Testen des Bits:

```
        AND     (HL)         ; alle Bits mit Maske verknuepfen
                             ; Z=0, wenn Bit gesetzt
```

Für das Setzen des Bits:

```
        OR      (HL)         ; alle Bits mit Maske verknuepfen
        LD      (HL),A       ; Bits zurueckschreiben
```

Für das Löschen des Bits:

```
        AND     (HL)         ; alle Bits mit Maske verknuepfen
        LD      (HL),A       ; Bits zurueckschreiben
```

Für das Invertieren des Bits:

```
        XOR     (HL)         ; alle Bits mit Maske verknuepfen
        LD      (HL),A       ; Bits zurueckschreiben
```

Die ständige Neuberechnung der Masken ist bei häufiger Ausführung der Bit-Operationen zu aufwendig. Wir legen deshalb zwei Byte-Felder mit jeweils 8 Elementen an, die unsere beiden Typen von Masken enthalten:

```
; Masken zum Setzen, Invertieren oder Testen eines Bits

SMASKE:  DEFB    00000001B    ; fuer Bit 0
         DEFB    00000010B    ; fuer Bit 1
         DEFB    00000100B    ; fuer Bit 2
         DEFB    00001000B    ; fuer Bit 3
         DEFB    00010000B    ; fuer Bit 4
         DEFB    00100000B    ; fuer Bit 5
         DEFB    01000000B    ; fuer Bit 6
         DEFB    10000000B    ; fuer Bit 7

; Masken zum Loeschen eines Bits

LMASKE:  DEFB    11111110B    ; fuer Bit 0
         DEFB    11111101B    ; fuer Bit 1
         DEFB    11111011B    ; fuer Bit 2
         DEFB    11110111B    ; fuer Bit 3
         DEFB    11101111B    ; fuer Bit 4
         DEFB    11011111B    ; fuer Bit 5
         DEFB    10111111B    ; fuer Bit 6
         DEFB    01111111B    ; fuer Bit 7
```

Der Zugriff auf die Masken geschieht nun mit Hilfe der Techniken, die wir für Byte-Felder gelernt haben. Wir bringen dazu b(i) nicht ins B-Register, sondern ins C-Register (obige Routine läßt sich leicht abändern). Die Adresse a(i), die sich im HL-Register befindet, sichern wir vorübergehend im DE-Register, um das HL-Register zunächst zur Adressierung der Maskenfelder benutzen zu können. Das Beschaffen der Maske zum Setzen des Bits erfolgt dann durch folgendes Programmstück (für die anderen Operationen entsprechend):

```
         EX      DE,HL        ; a(i) temporaer sichern
         LD      HL,SMASKE    ; Anfangsadresse des Maskenfelds
         LD      B,0          ; b(i) zu Wort erweitern
         ADD     HL,BC        ; Adresse der Maske berechnen
         LD      A,(HL)       ; Maske holen
         EX      DE,HL        ; a(i) ins HL-Register bringen
```

Durch einen kleinen Trick läßt sich bei Verwendung von Maskenfeldern die Routine DIVMOD nochmals verkürzen. Das Links-Rotieren war notwendig, damit wir die Bit-Adresse b(i) nicht spiegelverkehrt erhalten (zuerst würde ja das letzte Bit ankommen und schließlich in Bit 2 stehen). Durch Umordnen der Masken im Maskenfeld können wir aber auch mit dem spiegelverkehrten Wert von b(i) arbeiten. Dabei hilft uns folgende Zuordnungstabelle:

Tabelle 14.1. *Spiegelbilder der Bit-Adresse b(i)*

b(i)	Spiegelbild
0	0
1	4
2	2
3	6
4	1
5	5
6	3
7	7

Als Maskenfeld für das Löschen eines Bits ergibt sich dann:

```
; Masken zum Loeschen eines Bits

LMASKE:   DEFB    11111110B    ; fuer Bit 0
          DEFB    11101111B    ; fuer Bit 4
          DEFB    11111011B    ; fuer Bit 2
          DEFB    10111111B    ; fuer Bit 6
          DEFB    11111101B    ; fuer Bit 1
          DEFB    11011111B    ; fuer Bit 5
          DEFB    11110111B    ; fuer Bit 3
          DEFB    01111111B    ; fuer Bit 7
```

Die Routine DIVMOD ändern wir folgendermaßen:

```
DIVMOD:   LD      C,0          ; C-Register zur Aufnahme
                               ; des Spiegelbilds
                               ; von b(1) vorbereiten
          SRL     A            ; durch 2 dividieren
          RL      C            ; herausgefallenes Bit sichern
          SRL     A            ; durch 2 dividieren
          RL      C            ; herausgefallenes Bit sichern
          SRL     A            ; durch 2 dividieren
          RL      C            ; herausgefallenes Bit sichern
```

Da wir das C-Register zu Beginn gelöscht haben, ist nach jedem der Befehle RL C das Übertrag-Flag gelöscht. Statt des Befehls SRL A können wir deshalb anschließend den weniger aufwendigen Befehl RRA benutzen:

```
DIVMOD:   LD      C,0          ; C-Register zur Aufnahme
                               ; des Spiegelbilds
                               ; von b(i) vorbereiten
```

```
SRL     A       ; durch 2 dividieren
RL      C       ; herausgefallenes Bit sichern
RRA             ; durch 2 dividieren
RL      C       ; herausgefallenes Bit sichern
RRA             ; durch 2 dividieren
RL      C       ; herausgefallenes Bit sichern
```

Übungen

1. Erweitere die Routine ADRESS auf ganzzahlige Indizes, die in 2-Komplement-Darstellung im A-R gister angeliefert werden (andere Größen entsprechend anpassen).

2. Erweitere die Routine ADRESS auf vorzeichenlose ganzzahlige Indizes, die im HL-Register angeliefert werden (andere Größen entsprechend anpassen).

3. Denke Sie sich ein Deskriptorformat für eindimensionale Felder aus und schreiben Sie die Routine ADRESS so um, daß sie dieses verarbeiten kann; es wird dann nur die Adresse des Deskriptors und der Wert des Index übergeben. Baue eine Indexgrenzenüberwachung ein.

4. Schreibe eine Adressier-Routine für folgende Wort-Matrix:

$$\begin{pmatrix} 23177 & -32089 & 1423 & 17491 \\ -124 & 9678 & 0 & -12345 \\ 3356 & 0 & 7890 & 12978 \end{pmatrix}$$

5. Die Verständnisfrage: Realisiere ein Adressierverfahren für ein Feld, dessen Elemente jeweils 1 Bit (beziehungsweise 1 Nibble belegen).

14.3 Bearbeitung ganzer Felder

Häufig kommt es vor, daß nicht nur auf einzelne Elemente eines Felds zugegriffen wird, sondern daß alle Elemente eines Felds fortlaufend (beginnend mit dem kleinsten oder dem größten Index) bearbeitet werden müssen, oder zumindest ein zusammenhängender großer Indexbereich. Das Berechnen der einzelnen Adressen der Feldelemente mit Hilfe der Methoden von Unterkapitel 14.2 wäre in diesem Fall umständlich, da die innere Struktur des Felds dabei nicht gut ausgenutzt wird. Dieses Unterkapitel befaßt sich deshalb mit der Adressierung einer Menge von Feldelementen mit fortlaufenden Indizes aus einem großen zusammenhängenden Indexbereich. Da das allgemeine Problem sich nicht sehr von der fortlaufenden Adressierung *aller* Feldelemente unterscheidet, wollen wir nur zwei Typen von Adressierung untersuchen: Fortlaufende Adressierung aller Feldelemente mit aufsteigenden Indizes (mit absteigenden Indizes geht es ganz genauso!) und fortlaufende Adressierung von Feldelementen, beginnend beim kleinsten Index, bis zum Index eines Feldelements, das als erstes eine bestimmte -

jeweils vorzugebende - Bedingung erfüllt (Suche von rückwärts und/oder Suche bis zum n-ten Feldelement mit einer bestimmten Eigenschaft lassen sich daraus leicht ableiten).

Die einfachste Möglichkeit, alle Elemente eines Felds fortlaufend zu adressieren, besteht darin, eine Zählschleife zu konstruieren, deren Schleifenkörper sooft durchlaufen wird wie Indizes zu verarbeiten sind; der Schleifenkörper sorgt dann für die Adressierung der Feldelemente und für deren Bearbeitung. Die Adressierung erfolgt indirekt über ein Daten-Adreß-Register; wir wählen meist das HL-Register. Zu Beginn des Vorgangs wird das Adreß-Register auf die Adresse des ersten Feldelements gesetzt (wir wollen dies als bereits durchgeführt betrachten, wenn wir im folgenden Adressierungs-Routinen vorstellen); in jedem Durchlauf der Schleife wird dann dieser Zeiger auf das folgende Feldelement verschoben. Wir werden stets annehmen, daß die Zahl der zu bearbeitenden Elemente zwischen 1 und 256 liegt und wir deshalb eine automatische Zählschleife benutzen können (dies ist keine besonders wesentliche Einschränkung, da wir jederzeit auf selbstgesteuerte Zählschleifen umsteigen können); die Anzahl der Feldelemente erwarten wir stets im B-Register. Wir beginnen mit einem Byte-Feld, dessen Elemente wir negieren wollen:

```
NEGIER:  LD      A,(HL)     ; Feldelement beschaffen
         NEG                 ; Element negieren
         LD      (HL),A     ; Feldelement zurueckschreiben
         INC     HL         ; auf naechstes Element zeigen
         DJNZ    NEGIER     ; alle Feldelemente bearbeiten
```

Bei Byte-Feldern brauchen wir also nur den Zeiger zu inkrementieren. Bei Wort-Feldern genügt es, den Zeiger pro Schleifendurchlauf zweimal zu inkrementieren; ob das am Schleifenende oder während der Bearbeitung des Feldelements geschieht, ist prinzipiell egal. Wir sehen uns die Verdopplung aller Feldelemente eines Wort-Felds an. Die Adressierung ist ziemlich einfach, die Bearbeitung der Elemente dagegen kompliziert; dies kommt oft vor, und ist normalerweise ein gutes Zeichen (wenig organisatorischer Aufwand, engl. overhead).

```
DOPPEL:  LD      A,(HL)     ; niederwertiges Byte des
                            ; Feldelements holen
         ADD     A,A        ; niederwertiges Byte des
                            ; Feldelements verdoppeln
         LD      (HL),A     ; niederwertiges Byte des
                            ; Feldelements zurueckschreiben
         INC     HL         ; auf hoeherwertiges Byte zeigen,
                            ; Uebertrag-Flag bleibt intakt
         LD      A,(HL)     ; hoeherwertiges Byte des
                            ; Feldelements holen
         ADC     A,A        ; hoeherwertiges Byte des
                            ; Feldelements verdoppeln, dabei
                            ; Uebertrag beruecksichtigen
```

```
        LD      (HL),A      ; hoeherwertiges Byte des
                            ; Feldelements zurueckschreiben
        INC     HL          ; auf naechstes Element zeigen
        DJNZ    DOPPEL      ; alle Feldelemente bearbeiten
```

Der 8-Bit-Arithmetik-Befehl **ADC** (add carry) wirkt wie der 8-Bit-ADD-Befehl, addiert jedoch den Wert des Übertrag-Flags noch hinzu (vergleiche hierzu den 16-Bit-ADC-Befehl). Wir sehen hier bereits den Prototyp einer Addition beliebig großer ganzer Zahlen angedeutet; im Kapitel »ganze Zahlen« werden wir darauf noch genauer zu sprechen kommen.

Wir sehen bei unseren Programmen zunächst davon ab, daß die auszuführenden Operationen möglicherweise auf einen Fehler treffen (zum Beispiel Überlauf).

Der nächste Schwierigkeitsgrad ist erreicht, wenn die Länge der Feldelemente ein fortwährendes Inkrementieren des Zeigers unrationell werden läßt. Wir berechnen dann die Adresse des jeweils nächsten Feldelements durch Addition der Länge eines Elements zum aktuellen Zeiger, vorausgesetzt dieser ist durch die Bearbeitung des Feldelements noch nicht verändert worden. Als Beispiel laden wir den Wert 00H in das niederwertigste Byte jedes Feldelements, wobei die Elemente eine Länge von 16 Bytes haben sollen:

```
        LD      DE,16       ; Laenge eines Feldelements
NULLEN: LD      (HL),0      ; niederwertigstes Byte des
                            ; Feldelements Null setzen
        ADD     HL,DE       ; auf naechstes Element zeigen
        DJNZ    NULLEN      ; alle Feldelemente bearbeiten
```

An dieser Stelle bietet es sich an, über mehrdimensionale Felder zu sprechen. Wir betrachten als Beispiel ein zweidimensionales zeilenorientiertes Feld. Bearbeiten wir das Feld zeilenweise, so können wir alle Elemente ihrer Speicherungsreihenfolge nach verwerten; dasselbe gilt, wenn wir nur die Elemente einer bestimmten Zeile bearbeiten wollen. Sollen nur die Elemente einer bestimmten Spalte bearbeitet werden, so müssen wir die Adresse von Element zu Element um $(i_{o_2}-i_{u_2}+1) * $ »L« Byte fortschalten; dies können wir mittels der gerade gezeigten Technik bewerkstelligen. Der komplizierteste Fall liegt vor, wenn alle Elemente des Felds spaltenweise bearbeitet werden sollen; wir zeigen eine Adressierungs-Routine für ein zweidimensionales Byte-Feld mit 8 Zeilen und 12 Spalten (a_u wird im HL-Register erwartet):

```
; Datenbereich

BASIS:  DEFS    2           ; Hilfs-Speicherplatz fuer
                            ; Zeiger auf erstes Element
                            ; einer Spalte

; Programmbereich

DIM2:   LD      C,12        ; Anzahl der Spalten
        LD      DE,12       ; Anzahl der Spalten mal
                            ; Laenge eines Feldelements
```

```
SPALTE:   LD      (BASIS),HL  ; Zeiger auf erstes Element
                              ; einer Spalte sichern
          LD      B,8         ; Anzahl der Zeilen
NULLEN:   INC     (HL)        ; Feldelement um 1 erhoehen
          ADD     HL,DE       ; auf naechstes Element
                              ; derselben Spalte zeigen
          DJNZ    NULLEN      ; alle Feldelemente einer
                              ; Spalte bearbeiten
          LD      HL,(BASIS)  ; Zeiger auf erstes Element
                              ; der alten Spalte holen
          INC     HL          ; auf erstes Element der
                              ; naechsten Spalte zeigen
          DEC     C           ; restliche Anzahl der Spalten
                              ; berechnen
          JP      NZ,SPALTE   ; alle Spalten bearbeiten
```

Bisher haben wir stets nur ein Feld gleichzeitig bearbeitet. Häufig werden jedoch die Elemente zweier Felder verknüpft und die Ergebnisse in die Elemente des ersten Felds zurückgeschrieben. Wir wollen deshalb zwei Wort-Felder komponentenweise addieren und in das erste Feld zurückschreiben (Vektor-Addition). Dazu benötigen wir neben dem HL-Register ein weiteres Daten-Adreß-Register; wir wählen das DE-Register, da wir das B-Register zum Zählen benutzen:

```
VEKADD:   LD      A,(DE)      ; LSB eines Elements des
                              ; ersten Felds holen
          ADD     A,(HL)      ; LSB eines Elements des
                              ; zweiten Felds addieren
          LD      (DE),A      ; LSB der Summe zurueckschreiben
          INC     DE          ; auf MSB eines Elements
                              ; des ersten Felds zeigen
          INC     HL          ; auf MSB eines Elements
                              ; des zweiten Felds zeigen
          LD      A,(DE)      ; MSB eines Elements des
                              ; ersten Felds holen
          ADC     A,(HL)      ; MSB eines Elements des
                              ; zweiten Felds addieren,
                              ; dabei Uebertrag
                              ; beruecksichtigen
          LD      (DE),A      ; MSB der Summe zurueckschreiben
          INC     DE          ; auf naechstes Element
                              ; des ersten Felds zeigen
          INC     HL          ; auf naechstes Element
                              ; des zweiten Felds zeigen
          DJNZ    VEKADD      ; alle Feldelemente bearbeiten
```

Manchmal kommt es vor, daß wir das B-Register nicht als Zähler benutzen können, zum Beispiel wenn wir drei Felder simultan bearbeiten. Wir legen dann unsere Zählgröße temporär im Speicher ab. Folgende Subtraktion zweier Wort-Felder (Vektor-Differenz) mit Ablage des Ergebnisses in einem dritten Wort-Feld erläutert die Vorgehensweise (wir erwarten Zeiger auf das erste, zweite, dritte Feld in den Registern DE, HL, BC, und den Wert der Zählgröße im Speicherplatz ZAEHL):

```
; Datenbereich

ZAEHL:    DEFS      1          ; Hilfs-Speicherplatz
                               ; fuer Zaehlgroesse
; Programmbereich

VEKSUB:   LD        A,(DE)     ; LSB eines Elements des
                               ; ersten Felds holen
          SUB       (HL)       ; LSB eines Elements des
                               ; zweiten Felds subtrahieren
          LD        (BC),A     ; LSB der Summe in Element
                               ; des dritten Felds schreiben
          INC       DE         ; auf MSB eines Elements
                               ; des ersten Felds zeigen
          INC       HL         ; auf MSB eines Elements
                               ; des zweiten Felds zeigen
          INC       BC         ; auf MSB eines Elements
                               ; des dritten Felds zeigen
          LD        A,(DE)     ; MSB eines Elements des
                               ; ersten Felds holen
          SBC       A,(HL)     ; MSB eines Elements des
                               ; zweiten Felds subtrahieren,
                               ; dabei Uebertrag
                               ; beruecksichtigen
          LD        (BC),A     ; MSB der Summe in Element
                               ; des dritten Felds schreiben
          INC       DE         ; auf naechstes Element
                               ; des ersten Felds zeigen
          INC       HL         ; auf naechstes Element
                               ; des zweiten Felds zeigen
          INC       BC         ; auf naechstes Element
                               ; des dritten Felds zeigen
          LD        A,(ZAEHL)  ; Zaehlgroesse holen
          DEC       A          ; Zaehlgroesse dekrementieren
                               ; und auf Null testen
          LD        (ZAEHL),A  ; Zaehlgroesse wieder sichern
          JP        NZ,VEKSUB  ; alle Feldelemente bearbeiten
```

Der 8-Bit-Arithmetik-Befehl **SBC** (subtract carry) funktioniert wie der SUB-Befehl; er zieht jedoch vom Ergebnis noch den Wert des Übertrag-Flags ab. Einen SBC-Befehl für 16-Bit-Arithmetik haben wir im Kapitel »Worte« bereits besprochen.

Wir wollen die Bearbeitung von Feldern, deren Elemente je eine bestimmte Anzahl von Bytes belegen, abschließen mit einem Beispiel für die Suche nach einem Element mit einer vorgegebenen Eigenschaft. Dieses Problem wird am besten durch eine Zählschleife mit Abbruch gelöst; der Abbruch erfolgt, sobald das gewünschte Element gefunden ist. Wir suchen nun in einem Byte-Feld nach dem ersten Element größer als 99:

```
        LD      A,99        ; Testgroesse besetzen
FINDE:  CP      (HL)        ; Feldelement mit Testgroesse
                            ; vergleichen
        JP      C,GEFUND    ; Inhalt des Feldelements
                            ; groesser als Testgroesse,
                            ; Suche abbrechen
        INC     HL          ; auf naechstes Element zeigen
        DJNZ    FINDE       ; moeglicherweise alle
                            ; Feldelemente pruefen
NICHTG: NOP                 ; Feld enthaelt kein Element
                            ; mit gewuenschter Eigenschaft
```

Auch am Zustand des Übertrag-Flags können wir erkennen, ob die Suche erfolgreich war.

Nun kommen wir zu den Nibble-Feldern. Prinzipiell müssen wir dabei einerseits die Adressen fortlaufend fortschalten, andererseits innerhalb eines Bytes die Nibbles fortlaufend verarbeiten. Natürlich könnten wir die Verarbeitung der beiden Nibbles eines Bytes als eine zusammengehörige Operation ansehen; dies läßt sich mit den bisher besprochenen Methoden für die Bearbeitung von Byte-Feldern und den Bit-Manipulationstechniken leicht bewerkstelligen (bestimmte Operationen, wie das Null-Setzen aller Nibbles eines Nibble-Felds, können sogar direkt als Byte-Feld-Operationen ausgelegt werden). Andererseits können wir uns aber auch auf den Standpunkt stellen, daß bei komplizierten Operationen auf Nibbles die interne Struktur eines Bytes ausgenutzt werden sollte. Wir schachteln dann zwei Schleifen ineinander: die äußere Schleife ist eine Zählschleife, welche die fortlaufende Adressierung der Bytes gewährleistet; die innere Schleife ist ebenfalls eine Zählschleife, die dafür sorgt, daß der Reihe nach beide Nibbles eines Bytes bearbeitet werden. Das Beschaffen der Nibbles erfolgt für aufsteigende Indizes durch den Befehl RRD, für absteigende Indizes durch den Befehl RLD. Wir beginnen mit der aufsteigenden Bearbeitung eines Nibble-Felds, dessen Grenzen mit Byte-Grenzen übereinstimmen; dabei wollen wir den ersten Nibble mit dem Wert 00II ausfindig machen (au erwarten wir im HL-Register, die Anzahl der Bytes im B-Register):

```
        LD      A,0         ; der Test auf Null im RRD-Befehl
                            ; klappt nur, wenn der
                            ; hoeherwertige Nibble des
                            ; A-Registers geloescht ist
```

```
BYTE:   LD      C,2         ; 2 Nibbles pro Byte
NIBBLE: RRD                 ; Nibble ins A-Register holen
        JP      Z,GEFUND    ; Nibble ist Null,
                            ; Suche abbrechen
        DEC     C           ; restliche Anzahl von Nibbles
                            ; im Byte berechnen und auf
                            ; Null testen
        JP      NZ,NIBBLE   ; noch ein Nibble zu verarbeiten
        RRD                 ; Byte wiederherstellen
        INC     HL          ; auf naechstes Byte zeigen
        DJNZ    BYTE        ; alle Feldelemente pruefen
```

Wichtig sind bei diesem Algorithmus zwei Dinge:

1. Da wir das Nibble-Feld nicht beschädigen wollen, müssen wir durch *dreimaliges* Rotieren jedes Byte nach der Bearbeitung wiederherstellen.
2. Sobald wir einen Nibble mit der gewünschten Eigenschaft gefunden haben, wird die Suche abgebrochen; das gerade bearbeitete Byte müssen wir aber wiederherstellen, möglichst ohne die Zähler B und C zu zerstören, denn diese geben ja an, wo sich der gefundene Nibble befindet.

Um den zweiten Teil zu erledigen, fügen wir noch folgendes Programmstück an der Stelle ein, die nach geglückter Suche angesprungen wird:

```
GEFUND: LD      D,C         ; Nibble-Zaehler umspeichern
REPARA: RRD                 ; Byte reparieren
        DEC     D           ; Anzahl der noch
                            ; durchzufuehrenden Rotationen
                            ; berechnen
        JP      NZ,REPARA   ; genuegend Reparaturen
                            ; ausfuehren
```

Stimmen die Feldgrenzen nicht mit Byte-Grenzen überein, so muß ein »Vorlauf« beziehungsweise »Nachlauf« gemacht werden. Der Vorlauf besteht darin, daß wir den uns nicht interessierenden niederwertigen Nibble des ersten Bytes zwar holen, aber nicht testen, und dann mit reduziertem Nibble-Zähler in die innere Schleife einspringen (wir überspringen quasi den allerersten Test). Für den Nachlauf holen wir uns ohne Rotation das letzte Byte des Felds und maskieren den höherwertigen Nibble weg, weil das einfacher ist:

```
        LD      A,0         ; der Test auf Null im RRD-Befehl
                            ; klappt nur, wenn der
                            ; hoeherwertige Nibble des
                            ; A-Registers geloescht ist
VORLF:  RRD                 ; ersten Nibble nicht testen
```

```
        LD      C,1         ; Zaehler reduzieren
        JP      NIBBLE      ; in innere Schleife springen
BYTE:   LD      C,2         ; 2 Nibbles pro Byte
NIBBLE: RRD                 ; Nibble ins A-Register holen
        JP      Z,GEFUND    ; Nibble ist Null,
                            ; Suche abbrechen
        DEC     C           ; restliche Anzahl von Nibbles
                            ; im Byte berechnen und auf
                            ; Null testen
        JP      NZ,NIBBLE   ; noch ein Nibble zu verarbeiten
        RRD                 ; Byte wiederherstellen
        INC     HL          ; auf naechstes Byte zeigen
        DJNZ    BYTE        ; alle Feldelemente pruefen
NACHLF: LD      A,(HL)      ; letztes Byte des Felds holen
        AND     0FH         ; hoeherwertigen Nibble
                            ; wegmaskieren
        JP      Z,GEFUND    ; Suche war beim letzten
                            ; Nibble erfolgreich
```

Es kann bei der Bearbeitung von Nibble-Feldern also vorkommen, daß man weder Vorlauf noch Nachlauf benötigt, oder einen davon, oder schlimmstenfalls auch beide.

Die Bearbeitung von Nibble-Feldern mit absteigenden Indizes erfolgt ganz analog mit umgekehrter Rotationsrichtung.

Nun steht uns noch das tüfteligste Problem bevor: die Bearbeitung ganzer Bit-Felder! Dabei versuchen wir stets, (zunächst) nur Byte-Operationen durchzuführen.

Beispiel: Eine häufige Anwendung von Bit-Feldern sind Punktgraphiken; ein gesetztes Bit entspricht einem gezeichneten Punkt, ein gelöschtes Bit einem freien Bildpunkt. Natürlich kann man aus Punkten auch komplexere Gebilde wie Geraden oder Kreise zusammensetzen. Nun kann man auch zwei Bilder übereinanderlegen und dadurch zu einem neuen Bild verschmelzen. Dabei gibt es prinzipiell zwei Möglichkeiten: Überschreiben und Transparenz.

Beim Überschreiben wird überall dort ein Punkt eingetragen, wo sich in mindestens einem der beiden Bilder ein Punkt befindet. Wie man sich leicht überlegt, entspricht dies in unserer Bit-Darstellung der logischen Operation OR. Wir werden deshalb einfach die Bits mittels des OR-Befehls verknüpfen. Wir nehmen an, daß HL und DE Zeiger auf zwei Bilder sind, und daß in BC die Länge des Felds in Bytes steht (Bilder enthalten oft sehr viele Punkte); wir wollen das zweite Bild dem ersten überlagern:

```
UEBERL: LD      A,(DE)      ; 8 Bits des zweiten Felds
                            ; auf einmal holen
        OR      (HL)        ; simultan mit 8 Bits des
                            ; ersten Felds verknuepfen
        LD      (HL),A      ; resultierende Bits in erstes
                            ; Feld zurueckschreiben
```

```
        INC     HL          ; auf naechstes Byte des
                            ; ersten Felds zeigen
        INC     DE          ; auf naechstes Byte des
                            ; zweiten Felds zeigen
        DEC     BC          ; Anzahl der restlichen
                            ; Bytes berechnen
        LD      A,B         ; und auf
        OR      C           ; Null testen
        JP      NZ,UEBERL   ; alle Feldelemente bearbeiten
```

Im Transparent-Modus wird nur dort ein Punkt gesetzt, wo sich in genau einem der beiden Bilder ein Punkt befindet; dadurch werden Überschneidungen von Linien besser sichtbar. Hier verknüpfen wir die Bits durch den XOR-Befehl. Wir wollen nun zusätzlich annehmen, daß die untere Feldgrenze nicht mit einer Byte-Grenze zusammenfällt; die Bit-Adresse b_u des Felds übergeben wir im A-Register. Ein Programm für diese Aufgabe könnte so aussehen:

```
; Datenbereich

ZEIGER:  DEFS    2          ; Hilfs-Speicherplatz fuer
                            ; Zeiger auf erstes Feld
ZAEHL:   DEFS    2          ; Hilfs-Speicherplatz fuer
                            ; Byte-Zaehler
MASKEN:  DEFB    11111111B  ; Maske fuer bu=0
         DEFB    11111110B  ; Maske fuer bu=1
         DEFB    11111100B  ; Maske fuer bu=2
         DEFB    11111000B  ; Maske fuer bu=3
         DEFB    11110000B  ; Maske fuer bu=4
         DEFB    11100000B  ; Maske fuer bu=5
         DEFB    11000000B  ; Maske fuer bu=6
         DEFB    10000000B  ; Maske fuer bu=7

; Programmbereich

VORLF1:  LD      (ZEIGER),HL ; Zeiger auf erstes Feld sichern
         LD      (ZAEHL),BC  ; Byte-Zaehler sichern
         LD      HL,MASKEN   ; Anfangsadresse des
                             ; Maskenfelds laden
         LD      C,A         ; Bit-Adresse ins
         LD      B,0         ; BC-Register bringen
         ADD     HL,BC       ; Adresse der richtigen
                             ; Maske berechnen
         LD      A,(DE)      ; erstes Byte des
                             ; zweiten Felds holen
```

```
            AND     (HL)            ; nicht zum Bild gehoerige
                                    ; Bits loeschen
            LD      HL,(ZEIGER)     ; Zeiger auf erstes Feld
                                    ; wiederherstellen
            LD      BC,(ZAEHL)      ; Byte-Zaehler wiederherstellen
            JP      VORLF2          ; in Schleife einspringen
UEBERL:     LD      A,(DE)          ; 8 Bits des zweiten Felds
                                    ; auf einmal holen
VORLF2:     XOR     (HL)            ; simultan mit 8 Bits des
                                    ; ersten Felds verknuepfen
            LD      (HL),A          ; resultierende Bits in erstes
                                    ; Feld zurueckschreiben
            INC     HL              ; auf naechstes Byte des
                                    ; ersten Felds zeigen
            INC     DE              ; auf naechstes Byte des
                                    ; zweiten Felds zeigen
            DEC     BC              ; Anzahl der restlichen
                                    ; Bytes berechnen
            LD      A,B             ; und auf
            OR      C               ; Null testen
            JP      NZ,UEBERL       ; alle Feldelemente bearbeiten
```

Als letztes Problem wollen wir Adresse und Bit-Adresse des ersten gesetzten Bits eines Bit-Felds bestimmen; wir nehmen dabei an, daß die Feldgrenzen mit Byte-Grenzen zusammenfallen. Anstatt jedes Bit einzeln zu testen, bestimmen wir erst (im HL-Register) die Adresse des Bytes, in dem das gesuchte Bit liegt, und aus dem Byte selbst dann die Bit-Adresse (im B-Register). Ob die Suche erfolgreich war, wollen wir mittels des Übertrag-Flags anzeigen: Gesetztes Übertrag-Flag heißt erfolgreiche Suche, gelöschtes Übertrag-Flag bedeutet, daß alle Bits des Felds rückgesetzt sind.

```
BYTES:      LD      A,(HL)          ; 8 Feldelemente simultan
            OR      A               ; auf Null testen
            JP      NZ,GEFUND       ; mindestens ein Bit gesetzt
            INC     HL              ; auf naechstes Byte zeigen
            DEC     BC              ; Anzahl der restlichen
                                    ; Bytes berechnen
            LD      A,B             ; Anzahl der restlichen Bytes
            OR      C               ; auf Null testen
            JP      NZ,BYTES        ; alle Feldelemente pruefen
            JP      FERTIG          ; kein gesetztes Bit gefunden,
                                    ; Uebertrag-Flag ist durch
                                    ; den OR-Befehl geloescht
GEFUND:     LD      B,8             ; Anzahl der zu pruefenden Bits
```

```
BITS:     ADD     A,A          ; hoechstes Bit ins
                               ; Uebertrag-Flag bringen
          JP      C,KORREK     ; gesetztes Bit entdeckt,
                               ; Uebertrag-Flag ist gesetzt
          DJNZ    BITS         ; alle Bits pruefen
KORREK:   DEC     B            ; Korrektur fuer Bit-Adresse
FERTIG:   NOP                  ; gemeinsame Fortsetzungsstelle
```

Übungen

1. Ein Byte-Feld mit dem Indexbereich 0 – 255 soll als Ganzes bearbeitet werden; das i-te Element soll dabei den Wert i erhalten.

2. Schreibe ein Programm, das zwei Wort-Felder mit derselben Anzahl von Elementen vergleicht und die Adressen der ersten nicht übereinstimmenden Elemente liefert.

3. Schreibe ein Programm, das den letzten Nibble eines Nibble-Felds mit dem Wert 0FH sucht; dabei soll keine Feldgrenze mit einer Byte-Grenze übereinstimmen.

4. Schreibe ein Programm, das ein durch ein Bit-Feld realisiertes Bild invertiert; die Feldgrenzen fallen nicht unbedingt mit Byte-Grenzen zusammen.

14.4 Verschieben von Feldern

Wir haben bei der Bearbeitung ganzer Felder eine Operation ausgespart, die häufig vorkommt und für die es eine besonders elegante Lösung gibt: das Kopieren aller Feldelemente in ein anderes Feld. Da wir uns für die Inhalte der Feldelemente dabei gar nicht interessieren, sondern diese nur transportieren, können wir ein Feld einfach als zusammenhängenden Speicherbereich deuten. Unser Auftrag lautet also, eine bestimmte Anzahl von Bytes (beziehungsweise Nibbles oder Bits), beginnend bei einer vorgegebenen Adresse, in einen anderen Speicherbereich zu transportieren, dessen Adresse ebenfalls vorgegeben ist. Beide Deutungen sind gleichwertig, da wir einen Speicherbereich immer auch als Feld von Bytes, Nibbles oder Bits auffassen können. Grundsätzlich müssen wir drei verschiedene Situationen berücksichtigen:

1. Die beiden Speicherbereiche überlappen sich, und zwar am Ende des zu kopierenden Speicherbereichs.
2. Die beiden Speicherbereiche überlappen sich, und zwar am Anfang des zu kopierenden Speicherbereichs.
3. Die beiden Speicherbereiche sind disjunkt (sie überlappen sich nicht).

Im ersten Fall können wir nicht einfach die Feldelemente, beginnend mit der niedrigsten Adresse, in das zweite Feld kopieren, da wir dabei einige am Ende des ersten Felds gelegene

Elemente zerstören würden, bevor sie kopiert wurden. Beginnen wir jedoch mit dem Kopieren bei der höchsten Adresse, so wird das Feld korrekt kopiert.

Im zweiten Fall herrscht die umgekehrte Situation, wir müssen auf jeden Fall mit der niedrigsten Adresse beginnen.

Im dritten Fall tritt überhaupt kein Problem auf, wir können deshalb sowohl mit der niedrigsten wie auch mit der höchsten Adresse beginnen.

Wir wollen zunächst Felder kopieren, deren Elemente je ein ganzzahliges Vielfaches an Bytes belegen. Beim Transport braucht uns dann weder die genaue Länge eines Feldelements noch die Dimension des Felds bekannt zu sein. Wir benötigen nur folgende Angaben: Die Länge des Felds in Bytes und die Anfangs- oder Endadressen der beiden Felder. Anfangs- und Endadresse hängen über die Beziehung Endadresse = Anfangsadresse + Länge − 1 voneinander ab, so daß wir je nach Bedarf die benötigte Größe berechnen können.

Wir nehmen nun als erstes an, daß wir beim Kopieren mit der höchsten Adresse beginnen; dabei soll im HL-Register die Endadresse des ersten Felds stehen, im DE-Register die des zweiten Felds und im BC-Register die Länge der Felder in Bytes. Der Z80 hat einen sehr effizienten Befehl, dessen Ausführung das gesamte Problem auf einen Schlag erledigt: LDDR (load, decrement and repeat). Die Funktion des LDDR-Befehls kann wie folgt beschrieben werden:

wiederhole
(<DE>) <− <(<HL>)>
DE <− <DE> − 1
HL <− <HL> − 1
BC <− <BC> − 1
bis <BC> = 0

Dieser unscheinbare 2-Byte-Befehl ersetzt eine vollständige Zählschleife; alles was wir außerdem noch brauchen, ist eine vorhergehende Besetzung der drei Doppelregister BC, DE und HL.

Für das zweite Problem, den Kopiervorgang mit der niedrigsten Adresse zu beginnen, gibt es einen entsprechenden Befehl, LDIR (load, increment and repeat), mit der Formalisierung

wiederhole
(<DE>) <− <(<HL>)>
DE <− <DE> + 1
HL <− <HL> + 1
BC <− <BC> − 1
bis <BC> = 0

Wir nehmen nun an, daß wir nur die Länge der beiden Felder und ihre Anfangsadressen bekommen; bevor wir mit dem Kopieren beginnen können, müssen wir dann erst feststellen, ob dies von der niedrigsten oder der höchsten Adresse ab zu geschehen hat. Beginnen wir bei der niedrigsten Adresse, so können wir direkt die Anfangsadressen der Felder benutzen; ansonsten müssen wir die Endadressen mit Hilfe der Anfangsadressen und der Länge erst berechnen. Die Lösung sieht nun so aus:

wenn Anfangsadresse$_1$ < Anfangsadresse$_2$ <= Endadresse$_1$
dann kopiere rückwärts
sonst kopiere vorwärts

Im Flußdiagramm sieht der Ablauf dann folgendermaßen aus:

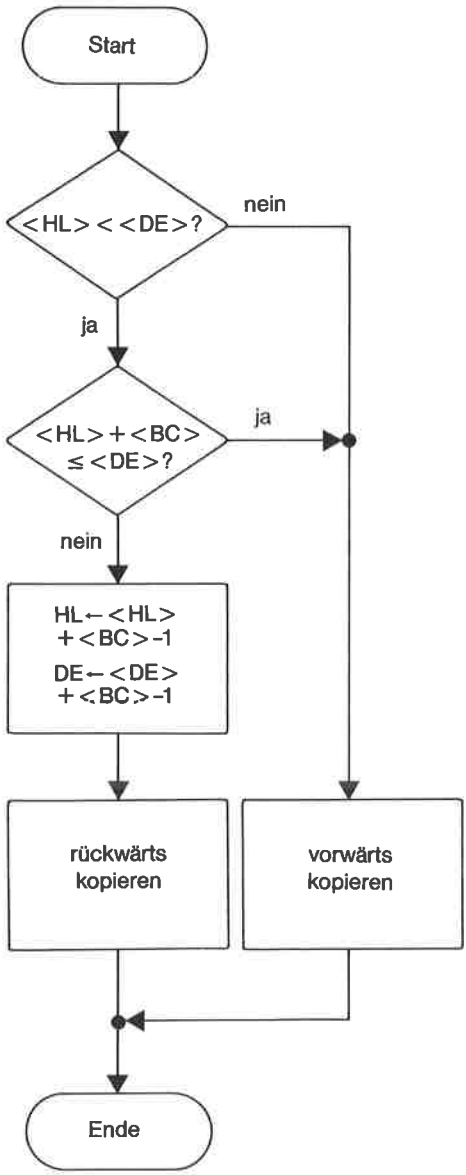


Bild 14.1. *Flußdiagramm: Verschieben von Feldern*

Das zugehörige Programm sieht so aus:

```
        OR      A           ; Uebertrag-Flag loeschen
        SBC     HL,DE       ; Anfangsadressen vergleichen
        JP      NC,VORW1    ; Anfangsadresse1 >=
                            ; Anfangsadresse2,
                            ; vorwaerts kopieren
        ADD     HL,DE       ; Anfangsadresse1 restaurieren
        ADD     HL,BC       ; Endadresse1 + 1 berechnen
        SCF                 ; Endadresse1 mit
        SBC     HL,DE       ; Anfangsadresse2 vergleichen
        JP      C,VORW2     ; Endadresse1 < Anfangsadresse2,
                            ; vorwaerts kopieren
        ADD     HL,DE       ; Endadresse1 berechnen
        EX      DE,HL       ; und temporaer sichern
        ADD     HL,BC       ; Endadresse2
        DEC     HL          ; berechnen
        EX      DE,HL       ; Endadressen tauschen
        LDDR                ; rueckwaerts kopieren
        JP      FERTIG      ; weiter an gemeinsamer
                            ; Fortsetzungsstelle
VORW1:  ADD     HL,DE       ; Anfangsadresse1 restaurieren
        JP      VORW        ; vorwaerts kopieren
VORW2:  ADC     HL,DE       ; Anfangsadresse1
        OR      A           ; wiederherstellen
        SBC     HL,BC       ;
VORW:   LDIR                ; vorwaerts kopieren
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Eine weitere Anwendungsmöglichkeit für den LDIR-Befehl liegt vor, wenn alle Elemente eines Felds mit demselben Wert initialisiert werden sollen. Wir bringen den Initialwert in das erste Element des Felds und kopieren ihn von dort in das zweite, vom zweiten in das dritte, und so weiter. Die richtigen Parameter für den Vorgang sind dabei: Anfangsadresse$_2$ = Anfangsadresse$_1$ + Länge eines Feldelements, Anzahl der Bytes = (Anzahl der Feldelemente − 1) * Länge eines Feldelements. Wir führen dies am Beispiel eines Wort-Felds vor, dessen Elemente mit dem Wert 0001H initialisiert werden sollen (HL ist Zeiger auf Anfang des Felds, BC enthält die Anzahl der Feldelemente):

```
        LD      (HL),01H    ; niederwertiges Byte des ersten
                            ; Feldelements initialisieren
        INC     HL          ; auf hoeherwertiges Byte des
                            ; ersten Feldelements zeigen
        LD      (HL),00H    ; hoeherwertiges Byte des ersten
                            ; Feldelements initialisieren
```

```
        LD      D,H     ; Zeiger
        LD      E,L     ; kopieren
        INC     DE      ; auf zweites Feldelement zeigen
        DEC     HL      ; auf erstes Feldelement zeigen
        DEC     BC      ; Anzahl der zu kopierenden
                        ; Feldelemente
        SLA     C       ; Anzahl der zu kopierenden
        RL      B       ; Bytes
        LDIR            ; kopieren
```

Wollen wir Kopiervorgänge abbrechen, sobald eine bestimmte Bedingung erreicht ist, zum Beispiel wenn das nächste zu kopierende Element einen bestimmten Wert besitzt, so steht uns die automatische Wiederholung des Kopierens eines Bytes im Weg. Der Z80 besitzt deshalb zwei Befehle, welche die Funktion des LDIR- und des LDDR-Befehls ohne Wiederholung realisieren: LDI (load and increment) und LDD (load and decrement). Beide Befehle löschen das Überlauf-Flag genau dann, wenn das BC-Register durch das Dekrementieren zu Null wurde; dies können wir benutzen, um eine Zählschleife mit Abbruch zu bauen. Als Beispiel kopieren wir die Elemente eines Byte-Felds in ein anderes Byte-Feld, solange diese größer als der Inhalt des A-Registers sind; die Anzahl der Feldelemente erwarten wir dabei im BC-Register, den Zeiger auf das erste Feldelement des ersten Felds im HL-Register, den entsprechenden Zeiger des zweiten Felds im DE-Register. Das Programm lautet damit:

```
KOPIE:  CP      (HL)        ; Feldelement mit Testgroesse
                            ; vergleichen
        JP      NC,FERTIG   ; Feldelement nicht groesser
                            ; als Testgroesse, abbrechen
        LDI                 ; Feldelement kopieren
        JP      PE,KOPIE    ; gesamtes Feld durchgehen
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Wir kommen nun zum Verschieben von Nibble- und Bit-Feldern. Stimmen die Nibble-Adressen (beziehungsweise Bit-Adressen) der beiden Felder überein, so ist es bei größeren Feldern am einfachsten, Anfangs- und End-Bytes (soweit sie nicht sowieso vollständig zum Feld gehören) separat zu bearbeiten (mit den Techniken aus dem Kapitel »Bit-Manipulationen«), alle übrigen Feldelemente aber Byte-weise zu kopieren.

Stimmen die Nibble-Adressen (beziehungsweise Bit-Adressen) dagegen nicht überein, so empfiehlt es sich, das Feld elementweise zu kopieren, und dabei Rotationsbefehle einzusetzen. Der Vorgang ist für Nibble-Felder und Bit-Felder sehr ähnlich, weshalb wir ihn erst einmal schematisch beschreiben:

1. Wir bauen eine Zählschleife auf, die bei jedem Durchlauf genau ein Feldelement transportiert; dabei wird zuerst durch Rotieren das Element aus dem ersten Feld geholt und dann durch Rotieren ins zweite Feld kopiert. Der Schleifenkörper besteht aus zwei Teilen: Holen des Feldelements und Abspeichern des Feldelements.

2. Beim Holen des Feldelements ist zu berücksichtigen, daß dieses Feld nicht verändert werden darf. Dies impliziert, daß wir beim Kopieren von Bit-Feldern mit zirkulären Rotationen arbeiten, und daß jedes Byte 8mal rotiert wird. Für Nibbles gibt es leider keine zirkulären Rotationsbefehle, so daß wir hier mit normalen Rotationen arbeiten müssen; jedes Byte wird deshalb 3mal rotiert. Außerdem müssen wir dabei den Nibble sichern, der beim nächsten Rotieren in das Byte gebracht werden soll (normalerweise ist dies der Nibble, der gerade herausrotiert wurde). Vereinbaren wir, daß das Feld durch das Kopieren zerstört werden darf, so entfällt das dritte Rotieren und das Sichern des Nibbles.
3. Beim Abspeichern des Feldelements wird das zweite Feld zerstört; alte Inhalte sind deshalb uninteressant und werden nirgendwo aufbewahrt. Das Einrotieren eines Feldelements geschieht durch gewöhnliches Rotieren; bei Bit-Feldern wird pro Byte 8mal rotiert, bei Nibble-Feldern 2mal.
4. Da die indirekte Adressierung für Rotations-Befehle nur durch das HL-Register zu bewerkstelligen ist, verwenden wir für beide Felder das HL-Register als Adreß-Register; nach Bearbeitung eines Feldelements müssen deshalb die Register DE und HL getauscht werden.
5. Das Inkrementieren oder Dekrementieren von Zeigern erfolgt immer beim Erreichen einer Byte-Grenze; deshalb müssen Zähler mitgeführt werden, welche die aktuellen Nibble-Adressen (beziehungsweise Bit-Adressen) wiedergeben.
6. Als Schleifenzähler dient das BC-Register (bei Bit-Feldern können damit bis zu 8 KByte adressiert werden).
7. Die Rotationsrichtung für aufsteigendes Kopieren ist rechts, für absteigendes Kopieren links.
8. Als Transporter für das Feldelement dient bei Bits das Übertrag-Flag, bei Nibbles das A-Register.
9. Fallen die Feldgrenzen nicht mit Byte-Grenzen zusammen, so müssen die Bytes am Anfang und/oder Ende der Felder zusätzlich behandelt werden.

Wir richten zwei Speicherplätze für die Zähler ein, bei Nibble-Feldern zusätzlich noch einen Speicherplatz für den ausrotierten Nibble.

Wir betrachten ein Programm zum aufsteigenden Verschieben von Nibble-Feldern; dabei nehmen wir an, daß die beiden Zählgrößen die Nibble-Adressen der ersten zu bearbeitenden Nibbles des jeweiligen Felds angeben. Wenn die untere Feldgrenze nicht mit einer Byte-Grenze übereinstimmt, so rotieren wir vor Beginn des eigentlichen Kopierens das erste Byte des ersten Felds nach rechts beziehungsweise vertauschen die beiden Nibbles des ersten Bytes des zweiten Felds. Wenn die obere Feldgrenze nicht mit einer Byte-Grenze übereinstimmt, so rotieren wir für das erste Feld noch zweimal nach rechts, während wir für das zweite Feld die beiden Nibbles tauschen müssen (viermal zirkulär rotieren):

```
; Datenbereich

ZAEHL1:   DEFS      1       ; Hilfs-Speicherplatz fuer
                            ; Zaehlgroesse des ersten Felds
ZAEHL2:   DEFS      1       ; Hilfs-Speicherplatz fuer
                            ; Zaehlgroesse des zweiten Felds
```

```
NIBBLE:   DEFS      1           ; Hilfs-Speicherplatz fuer
                                ; Nibble des ersten Felds

; Programmbereich

VORLF:    LD        A,(ZAEHL1)  ; Nibble-Adresse fuer
                                ; erstes Feld holen
          NEG                   ; Anzahl der Nibbles im ersten
          ADD       A,2         ; Byte berechnen
          LD        (ZAEHL1),A  ; Zaehlgroesse fuer erstes Feld
                                ; abspeichern
          DEC       A           ; auf volles Byte testen
          JP        NZ,NROT     ; volles Byte, nicht rotieren
          RRD                   ; nicht benoetigten Nibble
                                ; ausrotieren
          LD        (NIBBLE),A  ; und sichern
NROT:     LD        A,(ZAEHL2)  ; Nibble-Adresse fuer
                                ; zweites Feld holen
          NEG                   ; Anzahl der Nibbles im zweiten
          ADD       A,2         ; Byte berechnen
          LD        (ZAEHL2),A  ; Zaehlgroesse fuer zweites Feld
                                ; abspeichern
          DEC       A           ; auf volles Byte testen
          JP        NZ,KOPIE    ; volles Byte, nicht rotieren
          EX        DE,HL       ; Zeiger tauschen
          RLC       (HL)        ; nicht benoetigten Nibble so
          RLC       (HL)        ; rotieren, dass er
          RLC       (HL)        ; spaeter wieder
          RLC       (HL)        ; am alten Platz steht
          EX        DE,HL       ; Zeiger wieder tauschen
KOPIE:    LD        A,B         ; Anzahl der restlichen
          OR        C           ; Feldelemente auf Null testen
          JP        Z,NACHLF    ; alle Feldelemente kopiert
          LD        A,(NIBBLE)  ; einzurotierenden Nibble holen
          RRD                   ; alten Nibble einrotieren,
                                ; neuen Nibble beschaffen
          LD        (NIBBLE),A  ; neuen Nibble fuer naechsten
                                ; Durchgang sichern
          EX        DE,HL       ; Zeiger tauschen
          RRD                   ; neuen Nibble einrotieren
          EX        DE,HL       ; Zeiger wieder tauschen
          LD        A,(ZAEHL1)  ; Zaehler fuer erstes Feld holen
          DEC       A           ; neuen Zaehlwert berechnen
                                ; und auf Null testen
```

```
        JP      NZ,NNULL1 ; Zaehler noch nicht Null
        LD      A,(NIBBLE) ; einzurotierenden Nibble holen
        RRD               ; Byte wiederherstellen
        INC     HL        ; auf naechstes Byte des
                          ; ersten Felds zeigen
        LD      A,2       ; Zaehler neu laden fuer
                          ; 2 Nibbles
NNULL1: LD      (ZAEHL1),A ; Zaehler wieder abspeichern
        LD      A,(ZAEHL2) ; Zaehler fuer zweites Feld holen
        DEC     A         ; neuen Zaehlwert berechnen
        JP      NZ,NNULL2 ; Zaehler noch nicht Null
        INC     DE        ; auf naechstes Byte des
                          ; zweiten Felds zeigen
        LD      A,2       ; Zaehler neu laden fuer
                          ; 2 Nibbles
NNULL2: LD      (ZAEHL2),A ; Zaehler wieder abspeichern
        DEC     BC        ; restliche Anzahl von Nibbles
                          ; berechnen
        JP      KOPIE     ; alle Feldelemente kopieren
NACHLF: LD      A,(ZAEHL1) ; Zaehler fuer erstes Feld holen
        DEC     A         ; auf volles Byte testen
        JP      NZ,NROT2  ; volles Byte, nicht rotieren
        LD      A,(NIBBLE) ; einzurotierenden Nibble holen
        RLD               ; entspricht zwei Rechtsrotationen
NROT2:  LD      A,(ZAEHL2) ; Zaehler fuer zweites Feld holen
        DEC     A         ; auf volles Byte testen
        JP      NZ,FERTIG ; volles Byte, nicht tauschen
        EX      DE,HL     ; Zeiger tauschen
        RRC     (HL)      ; hoeherwertigen und
        RRC     (HL)      ; niederwertigen Nibble des
        RRC     (HL)      ; letzten Bytes des
        RRC     (HL)      ; Felds tauschen
        EX      DE,HL     ; Zeiger wieder tauschen
FERTIG: NOP               ; gemeinsame Fortsetzungsstelle
```

Das eben gezeigte Programm funktioniert allerdings nicht, wenn die Verschiebedistanz gerade einen Nibble beträgt; in diesem speziellen Fall muß das Programm folgendermaßen aussehen (wir nehmen der Einfachheit halber an, daß die untere Feldgrenze mit einer Byte-Grenze übereinstimmt und das Feld eine ungerade Anzahl von Nibbles enthält):

```
; Datenbereich

NIBBLE: DEFS    1         ; Hilfs-Speicherplatz fuer
                          ; ausrotierten Nibble
```

```
; Programmbereich

NIBB1:  INC     BC          ; Anzahl der zu
        SRL     B           ; bearbeitenden
        RR      C           ; Bytes berechnen
KOPIE:  RLD                 ; alten Nibble ausrotieren,
                            ; neuen Nibble einrotieren,
                            ; inneren Nibble verschieben
        LD      (NIBBLE),A  ; ausrotierten Nibble sichern
        INC     HL          ; auf naechstes Byte zeigen
        DEC     BC          ; restliche Zahl von Bytes
                            ; berechnen
        LD      A,B         ; und auf Null
        OR      C           ; testen
        LD      A,(NIBBLE)  ; ausrotierten Nibble
                            ; wieder holen
        JP      NZ,KOPIE    ; alle Feldelemente kopieren
```

Als letztes wollen wir ein Bit-Feld (um mindestens 8 Bits) verschieben, beginnend mit dem kleinsten Index, wobei die Feldgrenzen beliebig zu Byte-Grenzen liegen. Die Bit-Adressen der unteren Grenzen der Felder erwarten wir in den Variablen ZAEHL1 und ZAEHL2, die Adressen in den Registern HL und DE, die Anzahl der Bits der Felder im Register BC.

```
; Datenbereich

ZAEHL1: DEFS    1           ; Hilfs-Speicherplatz fuer
                            ; Zaehlgroesse des ersten Felds
ZAEHL2: DEFS    1           ; Hilfs-Speicherplatz fuer
                            ; Zaehlgroesse des zweiten Felds
BHILF:  DEFS    1           ; Hilfs-Speicherplatz fuer
                            ; Wert des B-Registers

; Programmbereich

VORLF:  LD      A,B         ; Wert des
        LD      (BHILF),A   ; B-Registers sichern
        LD      A,(ZAEHL1)  ; Zaehler fuer erstes Feld holen
        LD      B,A         ; Anzahl der auszurotierenden
                            ; Bits
        LD      A,8         ; Maximalzahl von Feldelementen
                            ; pro Byte
        SUB     B           ; Anzahl der Feldelemente im
                            ; ersten Byte berechnen
```

```
        LD      (ZAEHL1),A ; Zaehlgroesse abspeichern
        INC     B          ; abweisende Schleife modellieren
        JP      TEST1      ; in Schleife einspringen
VROT1:  RRC     (HL)       ; nicht benoetigtes Bit
                           ; wegrotieren
TEST1:  DJNZ    VROT1      ; alle nicht benoetigten
                           ; Bits wegrotieren
        EX      DE,HL      ; Zeiger tauschen
        LD      A,(ZAEHL2) ; Zaehler fuer zweites Feld holen
        LD      B,A        ; Anzahl der Zusatzrotationen
        LD      A,8        ; Maximalzahl von Feldelementen
                           ; pro Byte
        SUB     B          ; Anzahl der Feldelemente im
                           ; ersten Byte berechnen
        LD      (ZAEHL2),A ; Zaehlgroesse abspeichern
        INC     B          ; abweisende Schleife modellieren
        JP      TEST2      ; in Schleife einspringen
VROT2:  RRC     (HL)       ; Zusatzrotation ausfuehren
TEST2:  DJNZ    VROT2      ; alle Zusatzrotationen
                           ; ausfuehren
        EX      DE,HL      ; Zeiger wieder tauschen
        LD      A,(BHILF)  ; alten Wert des B-Registers
                           ; beschaffen
        LD      B,A        ; B-Register wiederherstellen
KOPIE:  RRC     (HL)       ; naechstes Bit beschaffen
        EX      DE,HL      ; Zeiger tauschen
        RR      (HL)       ; neues Bit einrotieren
        EX      DE,HL      ; Zeiger wieder tauschen
        LD      A,(ZAEHL1) ; Zaehler fuer erstes Feld holen
        DEC     A          ; neuen Zaehlwert berechnen
                           ; und auf Null testen
        JP      NZ,NNULL1  ; Zaehler noch nicht Null
        INC     HL         ; auf naechstes Byte des
                           ; ersten Felds zeigen
        LD      A,8        ; Zaehler neu laden fuer 8 Bits
NNULL1: LD      (ZAEHL1),A ; Zaehler wieder abspeichern
        LD      A,(ZAEHL2) ; Zaehler fuer zweites Feld holen
        DEC     A          ; neuen Zaehlwert berechnen
        JP      NZ,NNULL2  ; Zaehler noch nicht Null
        INC     DE         ; auf naechstes Byte des
                           ; zweiten Felds zeigen
        LD      A,8        ; Zaehler neu laden fuer 8 Bits
NNULL2: LD      (ZAEHL2),A ; Zaehler wieder abspeichern
```

```
        DEC     BC          ; restliche Anzahl von Bits
                            ; berechnen
        LD      A,B         ; und auf
        OR      C           ; Null testen
        JP      NZ,KOPIE    ; alle Feldelemente kopieren
NACHLF: LD      A,(ZAEHL1)  ; Zaehler fuer erstes Feld holen
        LD      B,A         ; Anzahl der noch noetigen
                            ; Rotationen
NROT1:  RRC     (HL)        ; Nachlauf-Rotation ausfuehren
        DJNZ    NROT1       ; alle Nachlauf-Rotationen
                            ; ausfuehren
        LD      A,(ZAEHL2)  ; Zaehler fuer zweites Feld holen
        LD      B,A         ; Anzahl der noch noetigen
                            ; Rotationen
        EX      DE,HL       ; Zeiger tauschen
NROT2:  RRC     (HL)        ; Nachlauf-Rotation ausfuehren
        DJNZ    NROT2       ; alle Nachlauf-Rotationen
                            ; ausfuehren
        EX      DE,HL       ; Zeiger wieder tauschen
```

Übungen

1. Der Wert des letzten Feldelements eines Byte-Felds soll in alle übrigen Feldelemente kopiert werden. Schreibe ein geeignetes Programm.

2. Kopiere die Elemente eines Byte-Felds von rückwärts in ein zweites Byte-Feld, bis eine Null erscheint.

3. Schreibe die letzten drei Programme für absteigendes Kopieren um.

4. Schreibe ein Programm, das die Verschiebung eines Bit-Felds um weniger als 8 Bits durchführt. Die Feldgrenzen fallen im allgemeinen dabei nicht mit Byte-Grenzen zusammen.

15
Zeichenketten

Ohne die Möglichkeit, Zeichenketten zu bearbeiten, ist ein Computer nur eine bessere Rechenmaschine!

Dies mag vielleicht etwas polemisch klingen. Aber jedes lauffähige Programm geht heutzutage mit Zeichenketten (engl. strings) um: bei der Eingabe werden Hinweise gegeben, welche Angaben benötigt werden und in welcher Form sie einzugeben sind; beim Ablauf eines Programms erhält der Benutzer Hinweise über aufgetretene Fehler oder den Fortgang der Arbeit; bei der Ausgabe von Daten werden diese durch begleitende Texte gekennzeichnet und erklärt. Die Zeit der rein numerischen Datenverarbeitung ist vorbei!

Es gibt mittlerweile sogar viele Programme zur Text- und Symbolverarbeitung, in denen numerische Daten nur eine untergeordnete Rolle spielen.

15.1 Implementierung von Zeichenketten

Es gibt so viele verschiedene Möglichkeiten, Zeichenketten darzustellen, daß wir hier gar nicht auf alle Varianten eingehen können; es sollen deshalb einige grundsätzliche Implementierungstechniken besprochen werden. In den folgenden Unterkapiteln befassen wir uns dann mit den Operationen auf Zeichenketten, wobei wir wieder nur eine Auswahl davon darstellen können.

Die einfachste Form einer Zeichenkette ist ein Feld von Zeichen, das eine feste Länge besitzt; die Länge ist a priori bekannt und kann deshalb in den Algorithmen als Konstante benutzt werden. Die Vereinbarung geschieht zum Beispiel mittels der Pseudo-Operation DEFM:

```
KETTE:    DEFM        'Hallo!'      ; Zeichenkette
                                    ; der festen Laenge 6
```

Ist der eigentliche Text kürzer als die Zeichenkette, so fügt man davor (rechtsbündiges Schreiben) oder dahinter (linksbündiges Schreiben) Leerzeichen ein.

Obwohl Zeichenketten fester Länge recht unflexibel sind (deshalb sind entsprechend wenig Operationen darauf sinnvoll), gibt es einen typischen Anwendungsbereich, nämlich Felder von Zeichenketten. Beim Aufbau von Feldern von Zeichenketten (oder bei Tabellen; diese werden wir in einem späteren Kapitel kennenlernen) können wir nur Elemente fester Länge benutzen, also entweder Zeichenketten fester Länge, oder statt der Zeichenkette selbst ihre Adresse (Deskriptor).

Eine sehr gebräuchliche Form ist die der Zeichenketten mit Längenangabe; dabei wird entweder direkt vor den Text, der nun eine variable Länge besitzen kann, oder in einem separaten Deskriptor die aktuelle Länge der Zeichenkette, das heißt die Anzahl der in ihr enthaltenen Zeichen, angegeben. Die Länge kann während der Laufzeit eines Programms variieren; sie kann unter Umständen sogar Null sein (leere Zeichenkette).

Wir zeigen zuerst die Implementierung einer Zeichenkette, deren Länge direkt vor dem Text steht:

```
KETTE:    DEFB      6           ; Laenge der Zeichenkette
          DEFM      'Hallo!'    ; eigentlicher Text
```

Manchmal begrenzt man die Länge auf 255 Zeichen, damit ein Byte für die Längenangabe genügt (sonst nimmt man ein Wort dafür her).

Nun ein Beispiel für eine Zeichenkette mit Deskriptor:

```
KETTE:    DEFB      6           ; Laenge der Zeichenkette
          DEFW      TEXT        ; Adresse des eigentlichen Texts
:
:
:
TEXT:     DEFM      'Hallo!'    ; Text steht irgendwo im Speicher
```

Die zweite Form benötigt mehr Speicherplatz, hat aber den Vorteil, daß der Deskriptor eine feste Länge hat und damit zum Beispiel an einer festen Stelle des Datenbereichs stehen kann, obwohl der Text seine Länge permanent ändert; insbesondere können solche Deskriptoren als Feldelemente verwendet werden.

Eine dritte Form markiert das Ende des Texts durch ein spezielles Zeichen, das im Text selbst dann natürlich nicht vorkommen darf. Meist handelt es sich um ein ASCII-Steuerzeichen, zum Beispiel 00H, 0DH, 1AH; falls die Textzeichen aus dem 7-Bit-ASCII-Code stammen, kann auch ein Zeichen mit einem Code größer als 7FH verwendet werden. Ein besonders raffinierter Trick, der keinen Speicherplatz verschenkt, ist die Methode, bei Verwendung des 7-Bit-ASCII-Codes das letzte Zeichen des Textes durch Setzen von Bit 7 zu markieren; zu dieser Technik gibt es sehr viele Varianten, die zum Beispiel in fest eingebauten Sprachinterpretern (vorwiegend BASIC) zur Anwendung kommen. Nachfolgend einige Beispiele von Zeichenketten mit Ende-Markierung:

```
KETTE1:   DEFM      'Hallo!'        ; eigentlicher Text
          DEFB      00H             ; Ende-Markierung
KETTE2:   DEFM      'Hallo!'        ; eigentlicher Text
          DEFB      0DH             ; Ende-Markierung
KETTE3:   DEFM      'Hallo!'        ; eigentlicher Text
          DEFB      1AH             ; Ende-Markierung
KETTE4:   DEFM      'Hallo!'        ; eigentlicher Text
          DEFB      0FFH            ; Ende-Markierung
KETTE5:   DEFM      'Hallo'         ; eigentlicher Text
                                    ; ohne letztes Zeichen
          DEFB      '!' + 80H       ; letztes Zeichen des Texts
                                    ; mit Ende-Markierung
```

Die vierte Form ist eine Kombination aus einer Zeichenkette fester Länge und einer Zeichenkette mit Ende-Markierung (vergleiche hierzu die Programmiersprache MODULA-2). Dabei wird eine Maximallänge für den Text vorgegeben; ist der aktuelle Text kürzer, so wird sein Ende wie bei der dritten Form markiert. Wir bringen hierzu zwei Beispiele für die Maximallänge 6:

```
KETTE1:   DEFM      'Hallo!'        ; Ende durch Laengenbegrenzung
KETTE2:   DEFM      'Ende'          ; eigentlicher Text
          DEFB      00H             ; Ende durch Markierung
```

Alle Formen von Zeichenketten kann man als Byte-Felder interpretieren; die Inhalte der Feldelemente haben dann je nach gewählter Form aber verschiedene Bedeutungen.

Übungen

1. Implementiere folgende Zeichenketten in den verschiedenen Formen der Darstellung (die spitzen Klammern kennzeichnen dabei Steuerzeichen des ASCII-Codes):

   Eingabe
   Guten Tag!
   Seite 4<CR><LF>

   (leere Zeichenkette)

   Verwende dabei 9 als konstante Länge, 00H beziehungsweise 0DH als Ende-Markierung und 10 als Längenbegrenzung. Gib jeweils die Größe des benötigten Speicherplatzes an!

15.2 Kopieren von Zeichenketten und Längenberechnungen

Da Zeichenketten im Prinzip Felder von Zeichen sind, können sie mit den Verfahren für Byte-Felder kopiert werden. Man muß allerdings dazu die Länge des Felds kennen. Bei Zeichenketten fester Länge ist diese bekannt, wir kopieren am besten mit Hilfe der Befehle LDDR oder LDIR. Ein Beispiel für das Kopieren einer Zeichenkette der festen Länge 6 wäre deshalb (das HL-Register zeigt auf den Anfang der Zeichenkette, das DE-Register auf den neuen Ablageort):

```
KOPIE:  LD      BC,6        ; Anzahl der zu transportierenden
                            ; Bytes
        LDIR                ; Transport durchfuehren
```

Beim Kopieren einer Zeichenkette mit Längenangabe müssen wir uns die aktuelle Länge der Zeichenkette aus dieser selbst holen, bevor wir den eigentlichen Kopiervorgang starten können. Für Zeichenketten, deren Längenangabe ein Byte benötigt, würde das so aussehen:

```
KOPIE:  LD      C,(HL)      ; Laenge des Texts holen
        LD      B,0         ; und zu Wort machen
        INC     BC          ; Laengenangabe mitzaehlen
        LDIR                ; Transport durchfuehren
```

Wichtig ist, daß die Längenangabe mitkopiert wird; deswegen müssen wir hier ein Byte mehr beim Transport angeben, als im Text der Zeichenkette enthalten ist.

Beim Kopieren einer Zeichenkette mit Ende-Markierung ist keine Länge bekannt; das Ende der Zeichenkette kann nur durch Abtasten aller Zeichen vom Anfang der Zeichenkette her gefunden werden. Wir benötigen deshalb einen gänzlich anderen Kopiermechanismus als bisher besprochen:

```
ENDE    EQU     0DH         ; Ende-Markierung, kann der
                            ;jeweiligen Form beliebig
                            ; angepasst werden

KOPIE:  LD      A,(HL)      ; Zeichen holen
        LD      (DE),A      ; Zeichen kopieren
        INC     HL          ; auf naechstes Zeichen zeigen
        INC     DE          ; auf naechsten freien
                            ; Speicherplatz zeigen
        CP      ENDE        ; kopiertes Zeichen mit
                            ; Ende-Markierung vergleichen
        JP      NZ,KOPIE    ; alle Zeichen einschliesslich
                            ; der Ende-Markierung kopieren
```

Auch hier ist es wichtig, die Ende-Markierung mitzukopieren.

Haben wir eine implizite Ende-Markierung durch Setzen von Bit 7 im letzten Zeichen des Texts gewählt, so vergleichen wir nicht auf ein bestimmtes Ende-Zeichen, sondern testen den Wert von Bit 7 des eben kopierten Zeichens (durch einen geeigneten logischen Befehl oder durch einen Rotationsbefehl):

```
KOPIE:  LD      A,(HL)      ; Zeichen holen
        LD      (DE),A      ; Zeichen kopieren
        INC     HL          ; auf naechstes Zeichen zeigen
        INC     DE          ; auf naechsten freien
                            ; Speicherplatz zeigen
        RLA                 ; Bit 7 des Zeichens testen
        JP      NC,KOPIE    ; alle Zeichen kopieren
```

Beim Kopieren einer Zeichenkette mit Längenbegrenzung und Ende-Markierung müssen wir zwei Testkriterien gleichzeitig benutzen: Überschreitung der Maximallänge oder Erreichen des Ende-Zeichens. Wir programmieren diese Aufgabe in Form einer Schleife mit Abbruch:

```
ENDE    EQU     0DH         ; Ende-Markierung, kann der
                            ; jeweiligen Form beliebig
                            ; angepasst werden
MAXL    EQU     80          ; Maximallaenge, kann
                            ; beliebig angepasst werden

        LD      B,MAXL      ; Maximalzahl von
                            ; zu kopierenden Zeichen
KOPIE:  LD      A,(HL)      ; Zeichen holen
        LD      (DE),A      ; Zeichen kopieren
        INC     HL          ; auf naechstes Zeichen zeigen
        INC     DE          ; auf naechsten freien
                            ; Speicherplatz zeigen
        CP      ENDE        ; kopiertes Zeichen mit
                            ; Ende-Markierung vergleichen
        JP      Z,WEITER    ; Ende, falls Ende-Markierung
                            ; gefunden und kopiert
        DJNZ    KOPIE       ; Ende, falls Maximallaenge
                            ; ueberschritten
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

Eine neben dem Kopieren ebenfalls häufig gewünschte Funktion ist das Bestimmen der Länge einer Zeichenkette, das ist nicht die Anzahl der Bytes, die sie belegt, sondern die Anzahl der Zeichenketten, die der eigentliche Text enthält. Für Zeichenketten fester Länge ist diese stets bekannt und braucht nicht bestimmt zu werden. Für Zeichenketten mit Längenangabe holt

man die Länge einfach aus dem ersten Byte (beziehungsweise den ersten beiden Bytes) der Zeichenkette. Für die übrigen Formen muß eine Suchschleife angelegt werden, zum Beispiel für Zeichenketten mit Ende-Markierung (das HL-Register soll wieder auf den Anfang der Zeichenkette zeigen):

```
ENDE     EQU     0DH          ; Ende-Markierung, kann der
                              ; jeweiligen Form beliebig
                              ; angepasst werden

         LD      A,ENDE       ; Ende-Markierung zum Vergleichen
         LD      B,0          ; Zaehler fuer Laenge
SUCHE:   CP      (HL)         ; Zeichen auf Ende-Markierung
                              ; testen
         JP      Z,FERTIG     ; Ende-Markierung gefunden
         INC     B            ; Zaehler erhoehen
         INC     HL           ; auf naechstes Zeichen zeigen
         JP      SUCHE        ; alle Zeichen ansehen
FERTIG:  NOP                  ; Fortsetzungsstelle
```

Für Zeichenketten mit impliziter Ende-Markierung läuft das Verfahren analog. Für Zeichenketten mit Längenbegrenzung und Ende-Markierung müssen wir den Algorithmus etwas modifizieren:

```
ENDE     EQU     0DH          ; Ende-Markierung, kann der
                              ; jeweiligen Form beliebig
                              ; angepasst werden
MAXL     EQU     80           ; Maximallaenge

         LD      A,ENDE       ; Ende-Markierung zum Vergleichen
         D       B,MAXL       ; Maximalzahl von Zeichen
         LD      C,0          ; Zaehler fuer Laenge
SUCHE:   CP      (HL)         ; Zeichen auf Ende-Markierung
                              ; testen
         JP      Z,FERTIG     ; Ende-Markierung gefunden
         INC     C            ; Zaehler erhoehen
         INC     HL           ; auf naechstes Zeichen zeigen
         DJNZ    SUCHE        ; alle Zeichen ansehen
FERTIG:  NOP                  ; gemeinsame Fortsetzungsstelle
```

Die Zeichenketten mit fester Länge besitzen gegenüber den Zeichenketten mit Längenangabe keine besonders gravierenden Unterschiede; die Algorithmen sehen deshalb auch fast gleich aus. Wir werden deshalb für den Rest dieses Kapitels nicht weiter auf Zeichenketten fester Länge eingehen. Die Algorithmen für Zeichenketten mit Längenbegrenzung und Ende-Mar-

kierung lassen sich leicht aus denen für Zeichenketten mit Längenangabe und für Zeichenketten mit Ende-Markierung ableiten. Die Zeichenketten mit impliziter Ende-Markierung unterscheiden sich nicht sehr stark von denen mit expliziter Ende-Markierung. Wir betrachten deshalb im folgenden nur noch zwei verschiedene Formen von Zeichenketten: Zeichenketten mit Längenangabe und Zeichenketten mit expliziter Ende-Markierung.

Übungen

1. Schreibe ein Programm für das Kopieren von Zeichenketten mit Längenangabe, wobei die Längenangabe ein Wort belegt.

2. Schreibe ein Programm für das Kopieren von Zeichenketten mit Ende-Markierung und Längenbegrenzung, wobei die maximale Länge auch größer als 255 sein kann.

3. Schreibe ein Programm, das die Länge einer Zeichenkette mit Ende-Markierung und Längenbegrenzung berechnet, wobei die maximale Länge auch größer als 255 sein darf.

15.3 Suchen in Zeichenketten und Vergleichsoperationen

Unter dem Begriff »Suchen in Zeichenketten« sind drei sehr stark zusammengehörige Operationen zusammengefaßt:

1. Stelle fest, ob ein bestimmtes Zeichen in einer Zeichenkette enthalten ist.
2. Liefere den Index des ersten Zeichens einer Zeichenkette, das mit einem bestimmten Zeichen übereinstimmt.
3. Liefere einen Zeiger auf das erste Zeichen einer Zeichenkette, das mit einem bestimmten Zeichen übereinstimmt.

Wir werden sehen, daß die dritte Aufgabe beim Behandeln der ersten Aufgabe quasi nebenbei erledigt wird.

Die Suche in Zeichenketten mit Längenangabe führen wir am besten mit dem Befehl **CPIR** (compare, increment and repeat) durch. Dieser Befehl vergleicht den Inhalt des A-Registers mit dem Inhalt der Speicherzelle, die durch das HL-Register indirekt adressiert wird; die Flags werden wie durch den Befehl CP (HL) gesetzt. Anschließend wird das HL-Register inkrementiert und das BC-Register – das als Zähler dient – dekrementiert. Falls der neue Wert des BC-Registers ungleich Null ist und beim Vergleichen das Null-Flag rückgesetzt wurde (das Zeichen der Zeichenkette war dann vom Testzeichen verschieden), so wird der Vorgang wiederholt. Der C-Befehl realisiert also eine Zählschleife (mit dem BC-Register als Zählgröße), die abgebrochen wird, sobald das gewünschte Zeichen gefunden wurde. Ob die Suche erfolgreich war, erkennen wir am Zustand des Null-Flags: gesetztes Null-Flag bedeutet gefunden, rückgesetztes Null-Flag bedeutet nicht gefunden. Dies löst die erste Aufgabe. Bei erfolgreicher Suche

brauchen wir nur noch das HL-Register zu dekrementieren, damit es auf das gefundene Zeichen zeigt; schon ist auch die dritte Aufgabe gelöst. Sind wir auch am Index des gefundenen Zeichens interessiert (Indizes bei Zeichenketten werden normalerweise ab 1 gezählt), so müssen wir den Wert des BC-Registers von der Länge der Zeichenkette abziehen. Wir können alle drei Aufgaben also durch folgendes Programm lösen, das als Längenangabe ein Wort erwartet (HL zeigt auf die Zeichenkette, A enthält das zu suchende Zeichen):

```
        LD      C,(HL)      ; Laenge der
        INC     HL          ; Zeichenkette
        LD      B,(HL)      ; holen und auf
        INC     HL          ; Text zeigen
        LD      E,C         ; Laenge
        LD      D,B         ; kopieren
        CPIR                ; Text absuchen
        SET     7,A         ; Zustand des
        JP      Z,NULL      ; Null-Flags in Bit 7
        RES     7,A         ; des A-Registers sichern
NULL:   DEC     HL          ; auf gefundenes Zeichen zeigen
        EX      DE,HL       ; Index
        OR      A           ; des gefundenen
        SBC     HL,BC       ; Zeichens
        EX      DE,HL       ; berechnen
```

Der Algorithmus funktioniert allerdings nur für nicht-leere Zeichenketten.

Den Zustand des Null-Flags haben wir in Bit 7 des A-Registers gesichert, HL zeigt auf das gesuchte Zeichen, DE enthält den Index.

Außer dem CPIR-Befehl gibt es noch den Befehl **CPDR** (compare, decrement and repeat), der das HL-Register in jedem Schritt dekrementiert, sonst aber genauso wirkt.

Schwieriger wird die Sache, wenn wir Zeichenketten mit Ende-Markierung absuchen. Hier müssen wir nach zwei Zeichen gleichzeitig sehen: dem Suchzeichen und dem Ende-Zeichen; eine automatische Wiederholung des Vergleichs durch den CPIR-Befehl kommt deshalb nicht in Frage. Wir bauen uns deshalb selbst eine Schleife; auf einen Schleifenzähler können wir dabei verzichten. Das HL-Register zeigt wieder auf die Zeichenkette, im C-Register befindet sich das Testzeichen; am Ende der Suche soll das Null-Flag uns wieder das Ergebnis der Suche mitteilen, das HL-Register auf das gefundene Zeichen zeigen und das DE-Register den Index beinhalten:

```
ENDE    EQU     0DH         ; Ende-Zeichen

        LD      DE,1        ; Index
SUCHE:  LD      A,(HL)      ; Zeichen holen
        CP      C           ; mit Testzeichen vergleichen
        JP      Z,WEITER    ; Zeichen gefunden
```

```
            INC     HL          ; auf naechstes Zeichen zeigen
            INC     DE          ; naechsten Index berechnen
            CP      ENDE        ; Zeichen mit Ende-Markierung
                                ; vergleichen
            JP      NZ,SUCHE    ; weitersuchen
            CP      C           ; Null-Flag loeschen
WEITER:     NOP                 ; gemeinsame Fortsetzungsstelle
```

Die Aufgabe kann man nun so modifizieren, daß nicht das erste Zeichen gesucht werden soll, das mit einem bestimmten Zeichen übereinstimmt, sondern das i-te Zeichen. Wir bauen in die Programme dann noch eine Zählschleife ein; die Initialisierung wird jeweils nur einmal durchgeführt. Wir zeigen dies zuerst für die Zeichenketten mit Ende-Markierung (die Anzahl i erwarten wir im B-Register):

```
ENDE        EQU     0DH         ; Ende-Zeichen

            LD      DE,1        ; Index
SUCHE:      LD      A,(HL)      ; Zeichen holen
            CP      C           ; mit Testzeichen vergleichen
            JP      NZ,NICHTG   ; Zeichen nicht gefunden
            DEC     B           ; restliche Anzahl errechnen
            JP      Z,WEITER    ; Zeichen i-mal gefunden
NICHTG:     INC     HL          ; auf naechstes Zeichen zeigen
            INC     DE          ; naechsten Index berechnen
            CP      ENDE        ; Zeichen mit Ende-Markierung
                                ; vergleichen
            JP      NZ,SUCHE    ; weitersuchen
            CP      C           ; Null-Flag loeschen
WEITER:     NOP                 ; gemeinsame Fortsetzungsstelle
```

Nun das entsprechende Programm für Zeichenketten mit Längenangabe (wir benötigen dabei zwei Hilfsvariablen):

```
; Datenbereich

HZEICH:     DEFS    1           ; Hilfs-Speicherplatz
                                ; fuer Testzeichen
HZAEHL:     DEFS    1           ; Hilfs-Speicherplatz
                                ; fuer Zaehlgroesse

; Programmbereich

            LD      (HZEICH),A  ; Testzeichen sichern
            LD      A,B         ; Zaehlgroesse
```

```
        LD      (HZAEHL),A      ; sichern
        LD      C,(HL)          ; Laenge der
        INC     HL              ; Zeichenkette
        LD      B,(HL)          ; holen und auf
        INC     HL              ; Text zeigen
        LD      E,C             ; Laenge
        LD      D,B             ; kopieren
SUCHE:  LD      A,(HZEICH)      ; Testzeichen holen
        CPIR                    ; Text absuchen
        JP      NZ,ENDE         ; Suche wird abgebrochen,
                                ; da Zeichenkette zu Ende
        LD      A,(HZAEHL)      ; Zaehlgroesse holen
        DEC     A               ; restliche Anzahl berechnen
        LD      (HZAEHL),A      ; Zaehlgroesse wieder sichern
        JP      NZ,SUCHE        ; Suche fortsetzen
NULL:   DEC     HL              ; auf gefundenes Zeichen zeigen
        EX      DE,HL           ; Index
        OR      A               ; des gefundenen
        SBC     HL,BC           ; Zeichens
        EX      DE,HL           ; berechnen
        XOR     A               ; Null-Flag setzen
ENDE:   NOP                     ; gemeinsame Fortsetzungsstelle
```

Ähnliche Probleme liegen vor, wenn in der laufenden Bearbeitung einer Zeichenkette das nächste Zeichen gesucht werden soll, das mit einem bestimmten Zeichen übereinstimmt.

Nun behandeln wir ein noch komplizierteres Problem, nämlich die Frage, ob eine Zeichenkette eine bestimmte andere Zeichenkette (deren Länge größer Null ist) enthält; diese nennen wir dann eine Teil-Zeichenkette (engl. substring). Auch hier könnten wir uns natürlich wieder den Index des ersten Zeichens der gefundenen Zeichenkette beschaffen; wir wollen aber aus Platzgründen darauf verzichten. Allerdings fällt durch das Verfahren der Zeiger auf eben dieses Zeichen automatisch mit an. Wir nehmen an, daß die beiden Zeichenketten in derselben Form vorliegen, daß HL auf die abzusuchende Zeichenkette zeigt und DE auf die Teil-Zeichenkette. Als erstes suchen wir in der Zeichenkette nach dem ersten Zeichen der Teil-Zeichenkette. Finden wir dieses nicht, so ist die Frage bereits negativ beantwortet. Ansonsten merken wir uns den Zeiger auf dieses Zeichen und vergleichen die folgenden Zeichen mit dem Rest der Teil-Zeichenkette. Stimmen sie überein, so ist die Frage positiv beantwortet. Wenn nicht, so holen wir den gesicherten Zeiger hervor und beginnen ab dieser Stelle von neuem nach dem ersten Zeichen zu suchen. Irgendwann bricht der Prozeß ab, weil die Teil-Zeichenkette gefunden wird, oder weil das Ende der Zeichenkette erreicht ist. Wir beginnen mit Zeichenketten mit Längenangabe:

```
; Datenbereich
ZEIGER: DEFS    2               ; Speicherplatz fuer Zeiger hinter
                                ; erstes gefundenes Zeichen
```

```
TEXT2:    DEFS    2          ; Speicherplatz fuer Zeiger auf
                             ; eigentlichen Text der
                             ; Teil-Zeichenkette
LAENGE:   DEFS    2          ; Speicherplatz fuer restliche
                             ; Laenge der Zeichenkette
LAENG2:   DEFS    2          ; Speicherplatz fuer Restlaenge
                             ; der Teil-Zeichenkette

;Programmbereich

          EX      DE,HL      ; Zeiger tauschen
          LD      C,(HL)     ; Laenge der
          INC     HL         ; Teil-Zeichenkette
          LD      B,(HL)     ; beschaffen und
          INC     HL         ; Restlaenge
          DEC     BC         ; in Variable
          LD      (LAENG2),BC ; abspeichern
          LD      (TEXT2),HL ; Zeiger auf eigentlichen Text
                             ; der Teil-Zeichenkette sichern
          EX      DE,HL      ; Zeiger wieder tauschen
          LD      C,(HL)     ; Laenge der
          INC     HL         ; Zeichenkette beschaffen und
          LD      B,(HL)     ; auf eigentlichen
          INC     HL         ; Text zeigen
SUCHE:    LD      A,(DE)     ; erstes Zeichen der
                             ; Test-Zeichenkette holen
          CPIR               ; in Zeichenkette danach suchen
          JP      NZ,ENDE    ; Ende der Zeichenkette
                             ; erreicht, Teil-Zeichenkette
                             ; nicht enthalten
          LD      (ZEIGER),HL ; Zeiger auf naechstes Zeichen
                             ; sichern
          LD      HL,(LAENG2) ; Restlaenge der
          OR      A          ; Teil-Zeichenkette mit
          SBC     HL,BC      ; Restlaenge der
                             ; Zeichenkette vergleichen
          JP      Z,WEITER   ; Laengen gleich?
          JP      NC,ENDE    ; Rest der Teil-Zeichenkette
                             ; laenger als Rest der
                             ; Zeichenkette, Teil-Zeichenkette
                             ; also nicht enthalten.
                             ; Null-Flag ist rueckgesetzt
WEITER:   LD      (LAENGE),BC ; restliche Laenge der
                             ; Zeichenkette sichern
```

```
            LD      BC,(LAENG2)  ; Restlaenge der
                                 ; Teil-Zeichenkette holen
            INC     DE           ; auf Rest der
                                 ; Teil-Zeichenkette zeigen
            LD      HL,(ZEIGER)  ; Zeiger auf Rest der
                                 ; Zeichenkette holen
VERGL:      LD      A,B          ; pruefen, ob
            OR      C            ; Teil-Zeichenkette
            JP      Z,GEFUND     ; abgearbeitet, wenn ja fertig
            LD      A,(DE)       ; Test-Zeichen holen
            CPI                  ; und mit Zeichen vergleichen
            INC     DE           ; auf naechstes
                                 ; Test-Zeichen zeigen
            JP      Z,VERGL      ; wenn Zeichen gleich
                                 ; Testzeichen, Vergleich
                                 ; fortsetzen
            LD      HL,(ZEIGER)  ; Zeiger auf Rest der
                                 ; Zeichenkette holen
            LD      DE,(TEXT2)   ; Zeiger auf Text der
                                 ; Teil-Zeichenkette holen
            LD      BC,(LAENGE)  ; Restlaenge der
                                 ; Zeichenkette holen
            JP      SUCHE        ; Suche von neuem beginnen
GEFUND:     LD      HL,(ZEIGER)  ; Zeiger auf erstes
            DEC     HL           ; gefundenes Zeichen berechnen
ENDE:       NOP                  ; gemeinsame Fortsetzungsstelle
```

Nun dasselbe für Zeichenketten mit Ende-Markierung:

```
; Datenbereich

ZEIGER:     DEFS    2            ; Speicherplatz fuer Zeiger hinter
                                 ; erstes gefundenes Zeichen
TEXT2:      DEFS    2            ; Speicherplatz fuer Zeiger auf
                                 ; Teil-Zeichenkette

; Programmbereich

ENDE        EQU     0DH          ; Ende-Markierung
            EX      DE,HL        ; Zeiger tauschen
            LD      (TEXT2),HL   ; Zeiger auf
                                 ; Teil-Zeichenkette sichern
SUCHE:      LD      A,(DE)       ; Zeichen holen
```

```
        CP      (HL)          ; mit Testzeichen vergleichen
        INC     DE            ; auf naechstes Zeichen zeigen
        JP      Z,ZGEF        ; Zeichen stimmen ueberein
        CP      ENDE          ; auf Ende der Zeichenkette
                              ; pruefen
        JP      NZ,SUCHE      ; Zeichenkette nicht zu Ende
NGEF:   CP      (HL)          ; Null-Flag loeschen
        JP      FERTIG        ; zum Fortsetzungspunkt
ZGEF:   LD      (ZEIGER),DE   ; Zeiger auf naechstes Zeichen
                              ; sichern
        INC     HL            ; auf Rest der
                              ; Teil-Zeichenkette zeigen
VERGL:  LD      A,(HL)        ; pruefen, ob
        CP      ENDE          ; Teil-Zeichenkette
        JP      Z,GEFUND      ; abgearbeitet, wenn ja fertig
        LD      A,(DE)        ; Zeichen holen
        CP      (HL)          ; mit Test-Zeichen vergleichen
        INC     DE            ; auf naechstes Zeichen zeigen
        INC     HL            ; auf naechstes
                              ; Test-Zeichen zeigen
        JP      Z,VERGL       ; wenn Zeichen gleich
                              ; Testzeichen, Vergleich
                              ; fortsetzen
        LD      DE,(ZEIGER)   ; Zeiger auf Rest der
                              ; Zeichenkette holen
        LD      HL,(TEXT2)    ; Zeiger auf
                              ; Teil-Zeichenkette holen
        JP      SUCHE         ; Suche von neuem beginnen
GEFUND: LD      HL,(ZEIGER)   ; Zeiger auf erstes
        DEC     HL            ; gefundenes Zeichen berechnen
FERTIG: NOP                   ; gemeinsame Fortsetzungsstelle
```

In den beiden Such-Algorithmen kam bereits die Vergleichsoperation zwischen Zeichenketten vor; dies läßt sich auf lexikalisches Vergleichen erweitern. So wie man auf den Zahlen die Relationen <, <=, >, >= , = und <> hat, kann man auch auf der Menge der Zeichenreihen eine Anordnung einführen; Gleichheit ist dabei eben Gleichheit der Texte, und die Kleiner-Relation geht in die Relation »steht lexikalisch vor« über. Die übrigen Relationen lassen sich aus diesen beiden ableiten oder durch Modifikation der Algorithmen direkt realisieren. Es gibt zwei Arten der lexikalischen Ordnung:

Die erste Art der lexikalischen Ordnung entspricht genau der Art, in der die Wörter eines Lexikons angeordnet sind. Es kommen bei der Bestimmung der Ordnung drei Regeln zur Anwendung:

1. Die leere Zeichenreihe (Zeichenreihe mit 0 Zeichen) steht vor allen nichtleeren Zeichenreihen.
2. Besitzt das erste Zeichen der einen Zeichenreihe einen kleineren ASCII-Code als das erste Zeichen der anderen Zeichenreihe, so steht die erste Zeichenreihe vor der zweiten (ist kleiner als die zweite). Beispiel: 'Affe' < 'Hund'.
3. Stimmt das erste Zeichen der einen Zeichenreihe mit dem ersten Zeichen der anderen Zeichenreihe überein, so steht die erste Zeichenreihe genau dann vor der zweiten Zeichenreihe, wenn der Rest der ersten Zeichenreihe (ohne das erste Zeichen) vor dem Rest der zweiten Zeichenreihe steht. Beispiel: 'Mars' < 'Mond'.

Die zweite Form der lexikalischen Ordnung ist sinnvoll, wenn wir es mit dem Vergleich von Zeichenreihen zu tun haben, die Zahlen darstellen; sie unterscheidet sich von der ersten Form nur im Wortlaut der ersten Regel:

1. Ist eine der Zeichenreihen kürzer als die andere, so steht die kürzere vor der längeren.

Wir beginnen nun mit dem Test auf Gleichheit für Zeichenketten mit Längenangabe (1 Byte). Dabei zeigt HL auf die erste Zeichenkette, DE auf die zweite Zeichenkette. Wenn die beiden Zeichenketten gleich sind, soll das Null-Flag gesetzt werden:

```
        LD      A,(DE)      ; Laenge der zweiten
                            ; Zeichenkette holen
        CP      (HL)        ; mit Laenge der ersten
                            ; Zeichenkette vergleichen
        JP      NZ,FERTIG   ; Laengen verschieden
        LD      B,(HL)      ; Laenge der beiden
                            ; Zeichenketten holen
        INC     B           ; Schleife abweisend machen
VERGL:  DEC     B           ; Restlaenge der Zeichenketten
                            ; berechnen und auf Null testen
        JP      Z,FERTIG    ; gesamte Laenge abgearbeitet,
                            ; Zeichenketten gleich
        INC     HL          ; auf naechstes Zeichen der
                            ; ersten Zeichenkette zeigen
        INC     DE          ; auf naechstes Zeichen der
                            ; zweiten Zeichenkette zeigen
        LD      A,(DE)      ; Zeichen der zweiten
                            ; Zeichenkette holen
        CP      (HL)        ; mit Zeichen der ersten
                            ; Zeichenkette vergleichen
        JP      Z,VERGL     ; Zeichen gleich, Rest der
                            ; Zeichenketten vergleichen
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Es folgt der Test auf Gleichheit für Zeichenketten mit Ende-Markierung:

```
ENDE    EQU     0DH         ; Ende-Markierung

VERGL:  LD      A,(DE)      ; Zeichen der ersten
                            ; Zeichenkette holen
        CP      (HL)        ; mit Zeichen der zweiten
                            ; Zeichenkette vergleichen
        JP      NZ,FERTIG   ; Zeichen verschieden, also
                            ; Zeichenketten verschieden
        INC     HL          ; auf naechstes Zeichen der
                            ; ersten Zeichenkette zeigen
        INC     DE          ; auf naechstes Zeichen der
                            ; zweiten Zeichenkette zeigen
        CP      ENDE        ; auf Ende der Zeichenketten
                            ; testen
        JP      NZ,VERGL    ; Zeichenketten nicht zu Ende
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Nun betrachten wir die Kleiner-Relation der ersten Art für Zeichenketten mit Längenangabe (das Übertrag-Flag wird gesetzt, falls die erste Zeichenkette kleiner als die zweite Zeichenkette ist):

```
        EX      DE,HL       ; Zeiger tauschen
        LD      C,0         ; C-Register zum Sichern des
                            ; Uebertrag-Flags vorbereiten
        LD      A,(DE)      ; Laenge der ersten
                            ; Zeichenkette holen
                B,(HL)      ; Laenge der zweiten
                            ; Zeichenkette holen
        CP      B           ; beide Laengen vergleichen
        JP      NC,KEINUE   ; Uebertrag-Flag nicht gesetzt
        INC     C           ; Uebertrag-Flag gesetzt
        LD      B,A         ; Minimum der Laengen
                            ; ins B-Register bringen
KEINUE: INC     B           ; Schleife abweisend machen
VERGL:  DJNZ    WEITER      ; Restlaenge berechnen und
                            ; auf Null testen
        RR      C           ; Uebertrag-Flag holen
        JP      FERTIG      ; zum Fortsetzungspunkt springen
WEITER: INC     DE          ; auf naechstes Zeichen der
                            ; ersten Zeichenkette zeigen
        INC     HL          ; auf naechstes Zeichen der
                            ; zweiten Zeichenkette zeigen
```

```
            LD      A,(DE)      ; Zeichen der ersten
                                ; Zeichenkette holen
            CP      (HL)        ; mit Zeichen der zweiten
                                ; Zeichenkette vergleichen
            JP      Z,VERGL     ; Zeichen stimmen ueberein,
                                ; weiter vergleichen
FERTIG:     NOP                 ; gemeinsamer Fortsetzungspunkt
```

Als nächstes betrachten wir die Kleiner-Relation der zweiten Art für Zeichenketten mit Längenangabe:

```
            EX      DE,HL       ; Zeiger tauschen
            LD      A,(DE)      ; Laenge der ersten
                                ; Zeichenkette holen
            CP      (HL)        ; mit Laenge der zweiten
                                ; Zeichenkette vergleichen
            JP      NZ,FERTIG   ; Laengen sind verschieden,
                                ; zum Fortsetzungspunkt springen
            LD      B,A         ; Laenge ins B-Register bringen
            INC     B           ; Schleife abweisend machen
VERGL:      DEC     B           ; Restlaenge berechnen und
                                ; auf Null testen
            JP      Z,FERTIG    ; Zeichenkette sind gleich,
                                ; zum Fortsetzungspunkt springen
            INC     DE          ; auf naechstes Zeichen der
                                ; ersten Zeichenkette zeigen
            INC     HL          ; auf naechstes Zeichen der
                                ; zweiten Zeichenkette zeigen
            LD      A,(DE)      ; Zeichen der ersten
                                ; Zeichenkette holen
            CP      (HL)        ; mit Zeichen der zweiten
                                ; Zeichenkette vergleichen
            JP      Z,VERGL     ; Zeichen stimmen ueberein,
                                ; weiter vergleichen
FERTIG:     NOP                 ; gemeinsamer Fortsetzungspunkt
```

Es folgt die Prüfung der Kleiner-Relation der ersten Art für Zeichenketten mit Ende-Markierung:

```
ENDE        EQU     0DH         ; Ende-Markierung

VERGL:      LD      A,(DE)      ; Zeichen der zweiten
                                ; Zeichenkette holen
```

```
            CP      ENDE        ; und auf Ende testen
            JP      Z,FERTIG    ; zweite Zeichenkette zu Ende,
                                ; erste Zeichenkette nicht
                                ; kleiner als zweite
            CP      (HL)        ; Zeichen der ersten Zeichenkette
                                ; mit Zeichen der zweiten
                                ; Zeichenkette vergleichen
            LD      A,(HL)      ; Zeichen der ersten
                                ; Zeichenkette holen
            INC     HL          ; auf naechstes Zeichen der
                                ; ersten Zeichenkette zeigen
            INC     DE          ; auf naechstes Zeichen der
                                ; zweiten Zeichenkette zeigen
            JP      Z,VERGL     ; beide Zeichen gleich,
                                ; weiter vergleichen
            JP      NC,WEITER   ; Zeichen der ersten Zeichenkette
                                ; kleiner als Zeichen der zweiten
                                ; Zeichenkette oder erste
                                ; Zeichenkette zu Ende, also
                                ; erste Zeichenkette kleiner
                                ; als zweite Zeichenkette
            CP      ENDE        ; Zeichen der ersten Zeichenkette
                                ; auf Ende testen
            SCF                 ; Annahme: Ende erreicht
            JP      Z,FERTIG    ; Annahme war richtig, erste
                                ; Zeichenkette kleiner als
                                ; zweite Zeichenkette
WEITER:     CCF                 ; Uebertrag-Flag richtig setzen
FERTIG:     NOP                 ; gemeinsame Fortsetzungsstelle
```

Nun fehlt uns noch die Kleiner-Relation der zweiten Art für Zeichenketten mit Ende-Markierung; wir führen diese Operation hier nicht aus, sondern stellen das Problem als Übungsaufgabe!

Übungen

1. Wie lassen sich die Relationen <=, >, >= und < > aus den Relationen = und < ableiten?

2. Schreibe ein Programm, das feststellt, ob ein bestimmtes Zeichen in einer Zeichenkette mit Längenbegrenzung und Ende-Markierung enthalten ist. Bedienen Sie sich dabei des Befehls CPI (compare and increment), dessen Definition Du aus dem Anhang entnehmen kannst.

3. Realisiere die Kleiner-Relation der zweiten Art für Zeichenketten mit Ende-Markierung.

15.4 Konkatenation und Ausschnitte von Zeichenketten

Bisher haben wir bei der Arbeit mit Zeichenketten diese niemals verändert; die Stärke der Zeichenketten-Verarbeitung liegt aber darin, aus Zeichenketten Teile herauszunehmen und zu neuen Zeichenketten zusammenzusetzen. Wir beginnen mit der Konkatenation von Zeichenketten, also dem Anfügen einer Zeichenkette an eine andere. Bei Zeichenketten mit Längenangabe müssen wir dabei die neue Länge als Summe der beiden alten Längen berechnen und die eigentlichen Texte aneinanderhängen. Wir arbeiten mit Längenangaben vom Typ »Wort«; dabei wollen wir annehmen, daß die neue Länge wieder als Wort dargestellt werden kann, und daß hinter der einen Zeichenkette auch Platz für die Zeichen der anderen Zeichenkette ist. Das DE-Register enthält einen Zeiger auf die Zeichenkette, an die angefügt werden soll, das HL-Register einen Zeiger auf die rechts davon anzufügende Zeichenkette:

```
; Datenbereich

ZEIGER:   DEFS      2          ; Hilfs-Speicherplatz fuer
                               ; Zeiger auf Text der rechten
                               ; Zeichenkette
LAENGE:   DEFS      2          ; Hilfs-Speicherplatz fuer
                               ; Laenge der linken Zeichenkette

; Programmbereich

          LD        C,(HL)     ; Laenge der
          INC       HL         ; rechten Zeichenkette
          LD        B,(HL)     ; holen und auf
          INC       HL         ; eigentlichen Text zeigen
          LD        A,B        ; auf leere Zeichenkette
          OR        C          ; pruefen
          JP        Z,FERTIG   ; rechte Zeichenkette leer,
                               ; nichts zu tun
          LD        (ZEIGER),HL ; Zeiger auf Text der rechten
                               ; Zeichenkette sichern
          EX        DE,HL      ; Zeiger tauschen
          LD        E,(HL)     ; Laenge der
          INC       HL         ; linken Zeichenkette
          LD        D,(HL)     ; holen
          EX        DE,HL      ; Zeiger wieder tauschen
          LD        (LAENGE),HL ; Laenge der linken
                               ; Zeichenkette sichern
          ADD       HL,BC      ; neue Laenge berechnen
          EX        DE,HL      ; Zeiger auf Gesamtlaenge
                               ; beschaffen
```

```
        LD      (HL),D          ; neue
        DEC     HL              ; Laenge
        LD      (HL),E          ; eintragen
        INC     HL              ; auf eigentlichen Text der
        INC     HL              ; linken Zeichenkette zeigen
        LD      DE,(LAENGE)     ; Laenge der linken
                                ; Zeichenkette holen
        ADD     HL,DE           ; hinter das Ende der linken
                                ; Zeichenkette zeigen
        EX      DE,HL           ; Zeiger sichern
        LD      HL,(ZEIGER)     ; Zeiger auf Text der rechten
                                ; Zeichenkette holen
        LDIR                    ; Text anfuegen
FERTIG: NOP                     ; gemeinsame Fortsetzungsstelle
```

Wesentlich einfacher gestaltet sich das Konkatenieren, wenn die Zeichenketten mit Ende-Markierung versehen sind (wir wollen auch hier voraussetzen, daß genügend Platz vorhanden ist):

```
ENDE    EQU     0DH             ; Ende-Markierung

SUCHE:  LD      A,(DE)          ; Zeichen der linken Zeichenkette
                                ; holen
        INC     DE              ; auf naechstes Zeichen der
                                ; linken Zeichenkette zeigen
        CP      ENDE            ; Zeichen mit Ende-Markierung
                                ; vergleichen
        JP      NZ,SUCHE        ; bis hinter das Ende der
                                ; linken Zeichenkette gehen
        DEC     DE              ; Ende-Markierung der linken
                                ; Zeichenkette wird
                                ; ueberschrieben
KOPIE:  LD      A,(HL)          ; Zeichen der rechten
                                ; Zeichenkette holen
        LD      (DE),A          ; Zeichen an linke
                                ; Zeichenkette anfuegen
        INC     HL              ; auf naechstes Zeichen der
                                ; rechten Zeichenkette zeigen
        INC     DE              ; auf naechstes Zeichen der
                                ; linken Zeichenkette zeigen
        CP      ENDE            ; Zeichen mit Ende-Markierung
                                ; vergleichen
        JP      NZ,KOPIE        ; weiter kopieren
```

Wir kommen nun zu den Funktionen, mit denen Teile einer Zeichenkette isoliert werden. Dabei sind prinzipiell zwei Zugänge zu unterscheiden: die Teil-Zeichenkette kann durch Indizes bezeichnet werden; sie kann aber auch durch Zeiger gegeben sein. Wir werden uns im folgenden jegliche Fehlerbehandlung sparen; es ist eine recht gute Übung, in einige der Programme eine solche einzubauen.

Ganz allgemein werden wir für alle folgenden Überlegungen voraussetzen, daß wir eine Zeichenkette bearbeiten wollen, ohne diese zu verändern; die Teil-Zeichenkette ist also eine eigenständige Zeichenkette, für die an einer vorgegebenen Stelle Speicherplatz freigehalten wird (die dabei auftretenden Probleme der Speicherplatzverwaltung sind keineswegs trivial, gehören aber eher in den Bereich der Systemprogrammierung). Wir wollen stets annehmen, daß das HL-Register auf die Zeichenkette zeigt, das DE-Register auf den für die Teil-Zeichenkette reservierten Speicherplatz; alle weiteren Parameter variieren je nach Funktion.

Die erste Funktion, die wir betrachten wollen, liefert die ersten n Zeichen einer Zeichenkette (von links gezählt). Die Zahl n soll im BC-Register stehen. Für Zeichenketten mit Längenangabe ist dies ein simpler Kopiervorgang mit vorgeschalteter Ablage der Länge der Teil-Zeichenkette:

```
        EX      DE,HL       ; Zeiger tauschen
        LD      (HL),C      ; Laenge der
        INC     HL          ; Teil-Zeichenkette ablegen
        LD      (HL),B      ; und auf eigentlichen
        INC     HL          ; Text zeigen
        EX      DE,HL       ; Zeiger wieder tauschen
        INC     HL          ; auf eigentlichen Text der
        INC     HL          ; Zeichenkette zeigen
        LDIR                ; Text kopieren
```

Bei Zeichenketten mit Ende-Markierung erfolgt erst das Kopieren und dann das Markieren des Textendes der Teil-Zeichenkette:

```
ENDE    EQU     0DH         ; Ende-Markierung

        LDIR                ; Text kopieren
        EX      DE,HL       ; Zeiger tauschen
        LD      (HL),ENDE   ; Ende der Teil-Zeichenkette
                            ; markieren
```

Ein ähnliches Problem liegt vor, wenn wir das Ende der Teil-Zeichenkette durch einen Zeiger bezeichnen (dieser zeigt normalerweise auf das nächste Zeichen hinter der gewünschten Zeichenkette); am einfachsten ist es, mit Hilfe des Zeigers die Anzahl der Zeichen, die kopiert werden sollen, zu berechnen, und dann die oben stehenden Algorithmen zu verwenden. Für Zeichenketten mit Längenangabe (als Wort) erfolgt diese Berechnung durch folgendes Programmstück (den Zeiger erwarten wir im BC-Register; wir berechnen also BC <– <BC> – <HL> – 2):

```
        DEC     BC
        DEC     BC
        LD      A,C
        SUB     L
        LD      C,A
        LD      A,B
        SBC     A,H
        LD      B,A
```

Für Zeichenketten mit Ende-Markierung lautet das entsprechende Programmstück (mit der Wirkung BC <– <BC> – <HL>):

```
        LD      A,C
        SUB     L
        LD      C,A
        LD      A,B
        SBC     A,H
        LD      B,A
```

Die nächste Operation liefert die Teil-Zeichenkette, die aus den letzten n Zeichen einer Zeichenkette besteht (n soll wieder im BC-Register stehen). Zuerst für Zeichenketten mit Längenangabe:

```
; Datenbereich

ZEIGER:  DEFS    2          ; Hilfs-Speicherplatz fuer
                            ; Zeiger auf letztes Zeichen
                            ; der Teil-Zeichenkette
; Programmbereich

        EX      DE,HL       ; Zeiger tauschen
        LD      (HL),C      ; Laenge
        INC     HL          ; der
        LD      (HL),B      ; Teil-Zeichenkette ablegen
        ADD     HL,BC       ; Zeiger auf letztes Zeichen
        LD      (ZEIGER),HL ; der Teil-Zeichenkette
                            ; berechnen und sichern
        EX      DE,HL       ; Zeiger wieder tauschen
        LD      E,(HL)      ; Laenge der
        INC     HL          ; Zeichenkette
        LD      D,(HL)      ; besorgen
        ADD     HL,DE       ; Zeiger auf letztes Zeichen
                            ; der Zeichenkette berechnen
```

```
        LD      DE,(ZEIGER)   ; Zeiger auf letztes Zeichen der
                              ; Teil-Zeichenkette restaurieren
        LDDR                  ; Text kopieren
```

Entsprechend für Zeichenketten mit Ende-Markierung:

```
ENDE    EQU     0DH           ; Ende-Markierung

        LD      A,ENDE        ; Ende-Markierung holen
SUCHE:  CP      (HL)          ; Zeichen mit
                              ; Ende-Markierung vergleichen
        INC     HL            ; auf naechstes Zeichen zeigen
        JP      NZ,SUCHE      ; bis zur Ende-Markierung suchen
KOPIE:  DEC     HL            ; auf Ende-Markierung der
                              ; Zeichenkette zeigen
        EX      DE,HL         ; Zeiger tauschen
        ADD     HL,BC         ; auf Ende-Markierung der
                              ; Teil-Zeichenkette zeigen
        EX      DE,HL         ; Zeiger wieder tauschen
        INC     BC            ; Ende-Markierung wird
                              ; ebenfalls kopiert
        LDDR                  ; Text kopieren
```

Ein ähnliches Problem ist, die Teil-Zeichenkette ab dem n-ten Zeichen (einschließlich) zu bilden. Wir geben exemplarisch nur die Variante für Zeichenketten mit Ende-Markierung an:

```
ENDE    EQU     0DH           ; Ende-Markierung

        ADD     HL,BC         ; Zeiger auf erstes zu
        DEC     HL            ; kopierendes Zeichen berechnen
KOPIE:  LD      A,(HL)        ; Zeichen holen
        LD      (DE),A        ; Zeichen kopieren
        INC     HL            ; auf naechstes Zeichen der
                              ; Zeichenkette zeigen
        INC     DE            ; auf naechstes Zeichen der
                              ; Teil-Zeichenkette zeigen
        CP      ENDE          ; Zeichen mit Ende-Markierung
                              ; vergleichen
        JP      NZ,KOPIE      ; bis einschliesslich der
                              ; Ende-Markierung kopieren
```

Zum Abschluß folgt die Beschaffung der Teil-Zeichenkette, die vom m-ten Zeichen (einschließlich) bis zum n-ten Zeichen (einschließlich) reicht. Wir zeigen die Variante für Zeichenketten mit Längenangabe (m steht im BC-Register, n in einer Variablen GRENZE):

```
;Datenbereich

GRENZE:   DEFS      2          ;End-Index n
ZEIGER:   DEFS      2          ;Hilfs-Speicherplatz fuer
                               ;Zeiger auf Teil-Zeichenkette

;Programmbereich

          ADD       HL,BC      ;Zeiger auf
          INC       HL         ;Teil-Zeichenkette
          LD        (ZEIGER),HL ;berechnen und sichern
          LD        HL,(GRENZE) ;oberen Index holen
          OR        A          ;Laenge der
          SBC       HL,BC      ;Teil-Zeichenkette
          INC       HL         ;berechnen
          LD        C,L        ;Laenge
          LD        B,H        ;kopieren
          LD        HL,(ZEIGER) ;Zeiger auf Teil-Zeichenkette
                               ;restaurieren
          EX        DE,HL      ;Zeiger tauschen
          LD        (HL),C     ;Laenge der
          INC       HL         ;Teil-Zeichenkette ablegen
          LD        (HL),B     ;und auf Text
          INC       HL         ;des Ablage-Bereichs zeigen
          EX        DE,HL      ;Zeiger wieder tauschen
          LDIR                 ;Text kopieren
```

Übungen

1. Schreibe Programme, die für Zeichenketten mit Längenangabe beziehungsweise Zeichenketten mit Ende-Markierung diejenige Teil-Zeichenkette bilden, deren Anfang durch einen Zeiger markiert wird und deren Ende mit dem Ende der Zeichenkette übereinstimmt.

2. Schreibe ein Programm, das die Teil-Zeichenkette ab dem n-ten Zeichen einer Zeichenkette mit Längenangabe bildet.

3. Schreibe ein Programm, das aus einer Zeichenkette mit Ende-Markierung die Teil-Zeichenkette ab dem m-ten Zeichen bis einschließlich des n-ten Zeichens isoliert.

15.5 Einfügen, Löschen und Ersetzen in Zeichenketten

Wir kommen nun zu Modifikationen von Zeichenketten, das heißt Änderungen an den Zeichenketten selbst. Eine wichtige Operation ist das Einfügen einer Zeichenkette in eine andere Zeichenkette an einer vorgegebenen Stelle. Wir nehmen nun an, daß die Einfügestelle durch einen Zeiger markiert ist, und zwar zeigt dieser auf den Speicherplatz, in den das erste Zeichen der einzufügenden Zeichenkette geschrieben werden soll. Wir haben es also mit drei Zeigern zu tun: HL zeigt auf die Zeichenkette, in die eingefügt werden soll, DE zeigt auf die einzufügende Zeichenkette, und BC zeigt auf die Stelle, an der eingefügt werden soll. Zuerst müssen wir für das Einfügen Platz in der Zeichenkette schaffen, indem wir den Text ab dem Zeiger BC nach hinten verschieben; dann können wir den Text der neuen Zeichenkette einfügen. Wir zeigen dies für Zeichenketten mit Längenangabe:

```
; Datenbereich

ZEIGDE:   DEFS    2          ; 1 + Inhalt von DE
ZEIGBC:   DEFS    2          ; Inhalt von BC
LAENG1:   DEFS    2          ; Laenge der Zeichenkette,
                             ; in die eingefuegt wird
LAENG2:   DEFS    2          ; Laenge der einzufuegenden
                             ; Zeichenkette

; Programmbereich

          LD      (ZEIGBC),BC ; Zeiger sichern
          EX      DE,HL       ; Zeiger tauschen
          LD      C,(HL)      ; Laenge der
          INC     HL          ; einzufuegenden
          LD      B,(HL)      ; Zeichenkette holen
          LD      (LAENG2),BC ; und sichern
          LD      A,B         ; Laenge auf
          OR      C           ; Null testen
          JP      Z,FERTIG    ; nichts einzufuegen
          LD      (ZEIGDE),HL ; 1 + alten Inhalt
                              ; von DE sichern
          EX      DE,HL       ; Zeiger tauschen
          LD      E,(HL)      ; Laenge der Zeichenkette,
          INC     HL          ; in die eingefuegt wird,
          LD      D,(HL)      ; holen,
          EX      DE,HL       ; verfuegbar machen
          LD      (LAENG1),HL ; und sichern
          ADD     HL,BC       ; neue Laenge berechnen
          EX      DE,HL       ; Zeiger holen
```

```
        LD      (HL),D      ; neue
        DEC     HL          ; Laenge
        LD      (HL),E      ; abspeichern
        LD      DE,(LAENG1) ; Zeiger auf Ende
        INC     HL          ; der Zeichenkette, in die
        ADD     HL,DE       ; eingefuegt wird, berechnen
        LD      D,H         ; und
        LD      E,L         ; sichern
        INC     HL          ; Laenge des zu verschiebenden
        LD      BC,(ZEIGBC) ; Textes der Zeichenkette, in
        OR      A           ; die eingefuegt wird,
        SBC     HL,BC       ; berechnen
        LD      B,H         ; und
        LD      C,L         ; sichern
        LD      HL,(LAENG2) ; Zeiger auf neues Ende der
        ADD     HL,DE       ; Zeichenkette berechnen
        EX      DE,HL       ; Zeiger tauschen
        LD      A,B         ; Laenge des zu verschiebenden
        OR      C           ; Speicherbereichs auf 0 testen
        JP      Z,EINFUE    ; nichts zu verschieben
        LDDR                ; Platz schaffen
EINFUE: LD      HL,(ZEIGDE) ; Zeiger auf Ende
        LD      BC,(LAENG2) ; der einzufuegenden
        ADD     HL,BC       ; Zeichenkette berechnen
        LDDR                ; Einfuegen
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Nach dieser anstrengenden Schiebeaktion verschnaufen wir kurz und widmen uns dann der zweiten wichtigen Operation, dem Löschen einer vorgegebenen Teil-Zeichenkette einer Zeichenkette. Hier nehmen wir an, daß DE auf die Teil-Zeichenkette zeigt und BC auf das nächste Zeichen hinter der zu löschenden Zeichenkette. Wir untersuchen die Variante für Zeichenketten mit Ende-Markierung (dazu brauchen wir keinen Zeiger auf die Zeichenkette, in der gelöscht wird):

```
ENDE    EQU     0DH         ; Ende-Markierung

KOPIE:  LD      A,(BC)      ; Zeichen holen
        LD      (DE),A      ; Zeichen kopieren
        INC     BC          ; beide Zeiger
        INC     DE          ; weiterbewegen
        CP      ENDE        ; Zeichen auf Ende testen
        JP      NZ,KOPIE    ; war nicht Ende, bis
                            ; einschliesslich Ende kopieren
```

Wir kommen nun zum letzten Problem dieses Kapitels, dem Umcodieren von Zeichen einer Zeichenkette. Wir gehen davon aus, daß zu jedem ASCII-Zeichen (mit 8 Bits) ein anderes ASCII-Zeichen (seine Codierung) definiert ist, welches das ursprüngliche Zeichen in der Zeichenkette ersetzen soll. Wir führen die Codierung an den Zeichen einer Zeichenkette mit Ende-Markierung aus, wobei BC auf den Anfang der Zeichenkette zeigt, DE auf den Anfang der Codierungstabelle:

```
ENDE     EQU     0DH          ; Ende-Markierung

CODIER:  LD      A,(BC)       ; Zeichen holen
         CP      ENDE         ; mit Ende-Markierung vergleichen
         JP      Z,FERTIG     ; Ende erreicht, fertig
         LD      L,A          ; Zeichen zu
         LD      H,0          ; Relativadresse machen
         ADD     HL,DE        ; Zeiger auf Codezeichen
                              ; berechnen
         LD      A,(HL)       ; Codezeichen holen
         LD      (BC),A       ; und Codierung ausfuehren
         INC     BC           ; auf naechstes Zeichen zeigen
         JP      CODIER       ; alle Zeichen codieren
FERTIG:  NOP                  ; Fortsetzungsstelle
```

Die Codierungstabelle ist ein Feld von Zeichen mit 256 Elementen, etwa folgendermaßen:

```
CODE:    DEFB    6BH          ; Codierung fuer 00H
         DEFB    9FH          ; Codierung fuer 01H
:
:
:
         DEFB    38H          ; Codierung fuer FFH
```

Viele Probleme mußten in diesem Kapitel ausgespart bleiben. Ich bedaure das, aber der Platz reicht einfach für ein noch genaueres Eingehen auf die Textverarbeitung nicht aus. Ich fordere deshalb Dich - den Leser - auf, in selbst ausgedachten Übungsaufgaben das Gelesene zu wiederholen und zu vertiefen.

Übungen

1. Schreibe ein Programm für das Einfügen in eine Zeichenkette mit Ende-Markierung.

2. Schreibe ein Programm, das eine Teil-Zeichenkette einer Zeichenkette mit Längenangabe löscht.

3. Schreibe ein Programm, das in einer Zeichenkette mit Ende-Markierung eine Teil-Zeichenkette durch eine andere Zeichenkette ersetzt. Alle Zeichenketten sollen durch Zeiger markiert sein, wo dies notwendig ist; die eingefügte Zeichenkette braucht nicht die Länge der gelöschten Teil-Zeichenkette zu haben.

16
Mengen

Eine *Menge* (engl. set) ist eine Zusammenfassung von Elementen; bei der Darstellung von Mengen in einem Computer läßt man normalerweise nur Elemente desselben Typs zu. Beispiele für Mengen wären eine Zahlenmenge (0, 1, ..., 255), eine Buchstabenmenge (A, B, ..., Z), eine Menge von Namen (Hans, Otto, Fritz, Franz, Heiner), eine Menge von Schachfiguren (König, Dame, Turm, Springer, Läufer, Bauer) oder eine Farbmenge (Rot, Gelb, Grün, Blau, Schwarz, Weiß). Die ersten beiden Mengen enthalten Elemente, die eine naheliegende Darstellung besitzen (Byte), die übrigen müssen wir erst sinnvoll codieren.

Man muß unterscheiden zwischen einer Menge und ihrer Repräsentation. Jede Menge kann nämlich - wie wir im folgenden sehen werden - mehrere Repräsentationen haben. Im Prinzip könnten wir Repräsentationen zulassen, in denen ein Element der dargestellten Menge auch mehrfach vorkommen darf. Wir wollen aber darauf verzichten, da dies zu einer Aufblühung der Datenobjekte führt und in manchen Fällen auch die Algorithmen komplizierter werden.

Auf den Elementen einer Menge können wir durch Angabe einer Ordnungsrelation eine Reihenfolge definieren. Bei der Verarbeitung von Mengenrepräsentationen bringt das meist jedoch erst dann einen Vorteil, wenn die Repräsentation diese Ordnung widerspiegelt. Wir werden von einer *geordneten Repräsentation* sprechen, wenn auf Menge und Repräsentation eine gemeinsame Ordnungsrelation definiert ist, sonst von einer *ungeordneten Repräsentation.*

Aus den Elementen einer Menge können wir neue, kleinere Mengen bilden, sogenannte Teilmengen (engl. subsets). Im folgenden geht es immer um die Darstellung dieser Teilmengen - kurz Mengen genannt - während wir die Grundmenge als fest und bekannt ansehen.

Für den Umgang mit Mengen gibt es einige Standardoperationen, aus denen wir kompliziertere Operationen aufbauen können. Diese Standardoperationen sollen nun zuerst einmal beschrieben werden:

Um überhaupt eine Menge aufbauen zu können, müssen wir in der Lage sein, ein einzelnes Element in eine schon bestehende Menge einzufügen; außerdem müssen wir die *leere Menge* aufbauen können, das ist eine Menge, in der sich kein Element befindet.

Weiterhin ist es sinnvoll, wenn wir prüfen können, ob ein bestimmtes Element sich in der Menge befindet.

Das Wegnehmen eines bestimmten Elements aus einer Menge kann auf Schwierigkeiten treffen, wenn dieses Element gar nicht in der Menge ist. Wir unterscheiden deshalb zwei Formen des Wegnehmens, nämlich das Wegnehmen eines vorhandenen Elements beziehungsweise den Versuch, ein bestimmtes Element wegzunehmen; auf jeden Fall befindet sich das Element hinterher nicht in der Menge.

Manchmal wollen wir wissen, wie viele Elemente sich in der Menge befinden; diese Anzahl nennt man die *Kardinalität* der Menge.

Zwei vorgelegte Mengen kann man auf Gleichheit testen; sie sind genau dann gleich, wenn sie dieselben Elemente enthalten. Ein spezieller Fall liegt vor, wenn wir feststellen wollen, ob eine Menge die leere Menge ist.

Eine Menge ist eine *Teilmenge* einer anderen Menge, wenn sie ausschließlich Elemente aus dieser enthält.

Zu einer Menge kann man das *Komplement* bilden, das ist diejenige Menge, die genau die Elemente der Grundmenge enthält, die nicht in der ursprünglichen Menge enthalten waren.

Die *Vereinigung* zweier Mengen enthält alle Elemente, die in mindestens einer der beiden Mengen vorkommen.

Ähnlich wie die Vereinigung funktioniert die *symmetrische Differenz* zweier Mengen; in ihr befinden sich nur diejenigen Elemente, die genau in einer der beiden Mengen vorkommen.

Die *Schnittmenge* zweier Mengen enthält nur die Elemente, die in beiden Mengen zugleich vorkommen.

Die (gewöhnliche) *Differenz* zweier Mengen – hier kommt es auf die Reihenfolge der beiden Mengen an – enthält genau die Elemente der ersten Menge, die nicht in der zweiten Menge liegen.

Keine Operation, sondern eine Klasse von Operationen, bildet die Anweisung, eine bestimmte Aktion auf jedem einzelnen Element einer Menge auszuführen (zum Beispiel das Ausdrucken der Elemente einer Menge, das Bilden der Summe einer Zahlenmenge, das Berechnen einer Farbmischung aus einer Menge von Farben).

Zu dieser Vielzahl von Operationen auf Mengen kommt noch eine große Freiheit in der Darstellung von Mengen. Wir unterscheiden primär zwei Formen der Darstellung: Wenn wir alle Elemente in irgendeiner Form auflisten, also ihre Werte sammeln, so sprechen wir von einer Darstellung durch Auflistung; in der zweiten Form ordnen wir jeder Menge ein spezielles Bit-Feld zu, ihren Inzidenzvektor.

16.1 Darstellung durch Auflistung

Für die Auflistung der Elemente einer Menge gibt es verschiedene Möglichkeiten. Zunächst kann man die Form eines Felds von Elementen wählen, wobei die Anzahl der Feldelemente die Kardinalität der Menge ist; diese muß also flexibel sein, weshalb ein Feld mit Deskriptor oder eine Tabelle (siehe das Kapitel »Tabellen«) angemessen ist. Es kann aber auch die Form einer Liste gewählt werden (siehe das Kapitel »Verzeigerte Datenstrukturen«). Bei all diesen

Darstellungen kann noch zwischen geordneter (zum Beispiel durch lexikalische Sortierung) oder ungeordneter Auflistung gewählt werden, wobei die geordnete meist Vorteile bringt. Nachfolgend einige Hinweise für das Umsetzen der Mengenoperationen in Algorithmen:

Die leere Menge wird als Feld der Länge Null, leere Tabelle oder leere Liste dargestellt; der Test auf leere Menge besteht dann im Prüfen der Länge. Die Kardinalität ist die Anzahl der Elemente der gewählten Struktur.

Beim Einfügen eines Elements muß (nach unserer oben getroffenen Vereinbarung) zuerst festgestellt werden, ob das Element schon vorhanden ist; in diesem Fall geschieht nichts. Ansonsten wird das neue Element angehängt (wenn ohne Anordnung gearbeitet wird) oder an der Stelle eingefügt, die der Anordnung entspricht (dies ist meist die Stelle, an der die Suche abgebrochen wurde). Die Kardinalität der Menge erhöht sich dabei um eins.

Beim Löschen eines Elements wird die gesamte Repräsentation durchsucht, falls keine Anordnung gegeben ist; ansonsten bricht die Suche meist früher ab. Ist das Element nicht in der Menge vorhanden, so bleibt die Operation ohne Wirkung; ansonsten wird das Element entfernt, wobei sich die Kardinalität um eins vermindert.

Die Prüfung, ob ein Element in der Menge enthalten ist, besteht aus einer Suche nach dem Element; bei ungeordneten Repräsentationen ist die Suche einfacher als bei geordneten, erstreckt sich aber dafür auf die gesamte Repräsentation.

Am deutlichsten macht sich der Unterschied zwischen geordneten und ungeordneten Repräsentationen beim Vergleich zweier Mengen bemerkbar. Bei geordneten Repräsentationen gehen wir die Elemente simultan in gemeinsamer Reihenfolge durch, bis zwei Elemente differieren (die Mengen sind dann nicht gleich) oder beide Repräsentationen abgearbeitet sind (die Mengen stimmen überein). Bei ungeordneten Repräsentationen müssen wir dagegen für jedes Element der einen Menge prüfen, ob es in der anderen Menge enthalten ist. In beiden Fällen empfiehlt es sich, zuerst die Kardinalitäten der beiden Mengen zu vergleichen. Bei gleichgroßen geordneten Repräsentationen mit n Elementen brauchen wir höchstens n Vergleiche (wenn die beiden Mengen übereinstimmen), bei ungeordneten Repräsentationen dagegen mindestens n Vergleiche, unter Umständen aber sogar n * (n+1) / 2 Vergleiche. Schon für kleines n (n=10) macht sich der Unterschied stark bemerkbar.

Für die Relation »Teilmenge von« gilt ein ähnliches Argument, da auch hier geprüft werden muß, ob jedes Element der einen Menge in der anderen enthalten ist.

Das Komplement einer Menge bilden wir, indem wir für jedes Element der Grundmenge feststellen, ob es in der Menge enthalten ist (dann wird es weggelassen) oder nicht (dann kommt es in die Komplement-Menge).

Die Vereinigung zweier Mengen bilden wir, indem wir sukzessive die Elemente der einen Menge zu der anderen Menge hinzugeben; wir müssen aber darauf achten, daß in der Repräsentation kein Element doppelt vorkommt. Bei der symmetrischen Differenz nehmen wir das Element sogar ganz weg, wenn es in beiden Mengen vorkommt.

Die Schnittmenge zweier Mengen finden wir, indem wir sukzessive für jedes Element der einen Menge feststellen, ob es auch in der anderen Menge vorkommt.

Bei der Bildung der Differenz zweier Mengen gehen wir sukzessive die Elemente der zweiten Menge durch und nehmen diese aus der ersten Menge fort, falls sie darin vorkommen.

Beim Ausführen einer bestimmten Aktion auf allen Elementen einer Menge kann diese für

geordnete und ungeordnete Repräsentationen gleichermaßen in beliebiger Reihenfolge durchlaufen werden.

Für die genaue Ausführung der Algorithmen sei auf die Kapitel »Tabellen« und »Verzeigerte Strukturen« verwiesen. Im allgemeinen sind geordnete Repräsentationen den ungeordneten vorzuziehen, wenn die Kardinalität der typischerweise auftretenden Mengen einen Wert in der Größenordnung von 10 oder mehr annimmt.

Übungen

1. Jede Zeichenkette, in der kein Zeichen doppelt vorkommt, kann man als Auflistung einer Menge von Zeichen ansehen. Schreibe Programme, die für ungeordnete Repräsentationen von Zeichenmengen (7 Bit ASCII) die Operationen »Gleich«, »Teilmenge von« und »Element von« realisieren. Versuche zu berechnen, wie viele Operationen mindestens, wie viele höchstens gebraucht werden.

2. Realisiere nun dieselben Operationen für geordnete Repräsentationen von Zeichenmengen. Vergleiche Mindest- und Höchstanzahl von Operationen mit denen der ersten Aufgabe.

16.2 Darstellung mit Hilfe von Inzidenzvektoren

Ein Inzidenzvektor einer Menge ist ein Bit-Feld. Jedes Element der Grundmenge korrespondiert zu genau einem Feldelement (wir haben es also mit einer geordneten Repräsentation zu tun). Das Element besitzt den Wert 1, wenn das Element in der Menge vorkommt, den Wert 0, wenn das Element nicht in der Menge vorkommt.

Inzidenzvektoren lassen sich recht einfach manipulieren; dadurch schleichen sich weniger Fehler in die Algorithmen ein, außerdem benötigt man meist weniger Zeit für die Bearbeitung. Ob Inzidenzvektoren verwendet werden können, hängt von der Kardinalität der Grundmenge ab; ist diese groß, so brauchen die Inzidenzvektoren sehr viel Speicherplatz, selbst wenn die auftretenden Mengen sehr klein sind (ein Inzidenzvektor benötigt immer denselben Speicherplatz, unabhängig von der Menge, die er darstellt). Als Beispiel betrachten wir die Menge der Zeichen (256 Elemente = 32 Byte Inzidenzvektor), für die Inzidenzvektoren noch tragbar sind, im Gegensatz zu Worten (2^{16} Elemente = 8 KByte Inzidenzvektor), wo ein einzelner Inzidenzvektor ein Achtel des gesamten Adreßraums des Z80 benötigen würde. Ein zweites Kriterium ist die Kardinalität der auftretenden Mengen; ist diese sehr klein (im Beispiel mit den Zeichen vielleicht 10 Zeichen), so ist die Verwendung von Inzidenzvektoren aufwendiger als die Auflistung.

Im folgenden wollen wir als Beispiel einer Grundmenge die Menge der Werte eines Nibbles wählen (0 bis 15); jeder Inzidenzvektor belegt dann ein Wort. Wir beginnen mit dem Herstellen der leeren Menge, wobei HL auf den Inzidenzvektor zeigt:

```
        XOR     A           ; Akkumulator loeschen
        LD      (HL),A      ; 8 Elemente loeschen
        INC     HL          ; auf die naechsten
                            ; 8 Elemente zeigen
        LD      (HL),A      ; 8 Elemente loeschen
```

Es folgt der Test auf das Vorliegen der leeren Menge. Wenn HL auf die leere Menge zeigt, soll das Null-Flag gesetzt werden:

```
        LD      A,(HL)      ; 8 Elemente laden
        INC     HL          ; auf die naechsten
                            ; 8 Elemente zeigen
        OR      (HL)        ; mit 8 Elementen verknüpfen
```

Als nächstes testen wir die Gleichheit zweier Mengen, auf die HL beziehungsweise DE zeigen. Bei Gleichheit soll das Null-Flag gesetzt werden:

```
        LD      A,(DE)      ; 8 Elemente holen
        CP      (HL)        ; 8 Elemente testen
        JP      NZ,FERTIG   ; Mengen sind verschieden
        INC     HL          ; auf die naechsten
        INC     DE          ; 8 Elemente zeigen
        LD      A,(DE)      ; 8 Elemente holen
        CP      (HL)        ; 8 Elemente testen
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Nun bilden wir das Komplement einer Menge. Das HL-Register zeigt auf die Menge, das DE-Register auf das Komplement:

```
        LD      A,(HL)      ; 8 Elemente holen
        CPL                 ; Komplement bilden
        LD      (DE),A      ; 8 Elemente abspeichern
        INC     HL          ; auf die naechsten
        INC     DE          ; 8 Elemente zeigen
        LD      A,(HL)      ; 8 Elemente holen
        CPL                 ; Komplement bilden
        LD      (DE),A      ; 8 Elemente abspeichern
```

Die Vereinigung zweier Mengen bilden wir durch die OR-Verknüpfung (HL beziehungsweise DE zeigen auf die beiden Mengen, BC auf die Vereinigung):

```
        LD      A,(DE)      ; 8 Elemente holen
        OR      (HL)        ; Vereinigung bilden
```

```
LD      (BC),A      ; 8 Elemente abspeichern
INC     HL          ; auf die
INC     DE          ; naechsten 8 Elemente
INC     BC          ; zeigen
LD      A,(DE)      ; 8 Elemente holen
OR      (HL)        ; Vereinigung bilden
LD      (BC),A      ; 8 Elemente abspeichern
```

Die symmetrische Differenz realisieren wir dagegen durch die XOR-Verknüpfung:

```
LD      A,(DE)      ; 8 Elemente holen
XOR     (HL)        ; symmetrische Differenz bilden
LD      (BC),A      ; 8 Elemente abspeichern
INC     HL          ; auf die
INC     DE          ; naechsten 8 Elemente
INC     BC          ; zeigen
LD      A,(DE)      ; 8 Elemente holen
XOR     (HL)        ; symmetrische Differenz bilden
LD      (BC),A      ; 8 Elemente abspeichern
```

Ebenso einfach erhalten wir durch die AND-Verknüpfung den Schnitt zweier Mengen:

```
LD      A,(DE)      ; 8 Elemente holen
AND     (HL)        ; Schnittmenge bilden
LD      (BC),A      ; 8 Elemente abspeichern
INC     HL          ; auf die
INC     DE          ; naechsten 8 Elemente
INC     BC          ; zeigen
LD      A,(DE)      ; 8 Elemente holen
AND     (HL)        ; Schnittmenge bilden
LD      (BC),A      ; 8 Elemente abspeichern
```

Bei der Bildung der Differenz einer Menge A mit einer anderen Menge B (A – B) beachten wir, daß die resultierende Menge auch als Schnittmenge von A mit dem Komplement von B dargestellt werden kann. Zeigt HL auf die Menge A, DE auf die Menge B, BC auf die Differenzmenge, so lautet das Programm:

```
LD      A,(DE)      ; 8 Elemente holen
CPL                 ; Komplement bilden
AND     (HL)        ; Schnittmenge bilden
LD      (BC),A      ; 8 Elemente abspeichern
INC     HL          ; auf die
INC     DE          ; naechsten 8 Elemente
```

```
        INC     BC          ; zeigen
        LD      A,(DE)      ; 8 Elemente holen
        CPL                 ; Komplement bilden
        AND     (HL)        ; Schnittmenge bilden
        LD      (BC),A      ; 8 Elemente abspeichern
```

Wollen wir feststellen, ob eine Menge A Teilmenge einer Menge B ist, so können wir testen, ob der Schnitt von A und B mit A übereinstimmt. HL zeigt auf die Menge A, DE auf die Menge B; ist A Teilmenge von B, so soll das Null-Flag gesetzt werden:

```
        LD      A,(DE)      ; 8 Elemente holen
        AND     (HL)        ; Schnittmenge bilden
        CP      (HL)        ; 8 Elemente vergleichen
        JP      NZ,FERTIG   ; A nicht Teilmenge von B
        INC     HL          ; auf die naechsten
        INC     DE          ; 8 Elemente zeigen
        LD      A,(DE)      ; 8 Elemente holen
        AND     (HL)        ; Schnittmenge bilden
        CP      (HL)        ; 8 Elemente vergleichen
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Für die restlichen Operationen bleibt uns nichts anderes übrig, als die einzelnen Elemente der Menge, das heißt die Elemente des Bit-Felds zu bearbeiten. Für das Hinzufügen eines Elements müssen wir das entsprechende Bit setzen, für das Wegnehmen müssen wir es löschen; um festzustellen, ob ein bestimmtes Element in der Menge enthalten ist, testen wir das entsprechende Bit. Alle drei Techniken sind in Kapitel 14.2 genau beschrieben.

Zur Berechnung der Kardinalität einer Menge rotieren wir die von ihr belegten Bytes und zählen die herausgeschobenen Bits, die den Wert 1 tragen (HL zeigt auf die Menge, A enthält anschließend die Kardinalität):

```
        XOR     A           ; Akkumulator loeschen
        LD      C,2         ; Anzahl der Bytes
BYTES:  LD      B,8         ; Anzahl der Bits pro Byte
BITS:   RRC     (HL)        ; Indikator herausschieben
        ADC     A,0         ; und aufaddieren
        DJNZ    BITS        ; alle Bits einbeziehen
        INC     HL          ; auf die naechsten
                            ; 8 Elemente zeigen
        DEC     C           ; restliche Anzahl von Bytes
        JP      NZ,BYTES    ; alle Bytes einbeziehen
```

Wollen wir auf jedem Element eine Aktion durchführen, so gehen wir ähnlich vor; wir schieben wieder das entsprechende Bit heraus, testen es und führen die Aktion durch, wenn es gesetzt ist. Als Beispiel bilden wir die Summe der Elemente einer Menge von Zahlen:

```
        XOR     A           ; Akkumulator loeschen
        LD      E,A         ; naechstes Element der Menge
        LD      C,2         ; Anzahl der Bytes
BYTES:  LD      B,8         ; Anzahl der Bits pro Byte
BITS:   RRC     (HL)        ; Element herausschieben
        JP      NC,WEITER   ; Element nicht in der Menge
        ADD     A,E         ; Element aufaddieren
WEITER: INC     E           ; naechstes Element der Menge
        DJNZ    BITS        ; alle Bits einbeziehen
        INC     HL          ; auf die naechsten
                            ; 8 Elemente zeigen
        DEC     C           ; restliche Anzahl von Bytes
        JP      NZ,BYTES    ; alle Bytes einbeziehen
```

Übungen

1. Programmiere die Operationen Vereinigung, Schnitt und Differenz für die Menge der kleinen Buchstaben.

2. Programmiere die Operationen Komplement und symmetrische Differenz für die Menge der ASCII-Zeichen (7 Bits).

3. Zum Tüfteln: In der elementaren Farbenlehre kommt man mit den Farben Rot, Gelb, Blau, Grün, Orange, Violett, Weiß und Schwarz aus. Schreibe ein Programm, das zu einer Menge von Farben die Resultatfarbe der additiven beziehungsweise subtraktiven Mischung bestimmt (Näheres über die Farbenlehre kann jedem besseren Lexikon entnommen werden).

17
Verbunde

Ein *Verbund* (engl. record) ist eine Zusammenfassung von Daten (möglicherweise verschiedenen Typs) zu einem Datensatz. Die Anzahl der Komponenten eines Verbunds und ihre Typen liegen durch die Definition der Datenstruktur fest. Anders als bei Feldern kann man die Adressen der Komponenten eines Verbunds nicht durch eine einfache Funktion angeben.

Die einfachste Form eines Verbunds ist der *ungepackte Verbund*. Dabei beginnen alle Komponenten – unabhängig von ihrem tatsächlichen Speicherbedarf – an einer Byte-Grenze; es wird dadurch möglicherweise Speicherplatz verschwendet (wenn nämlich eine Komponente eine nicht durch 8 teilbare Anzahl von Bits benötigt). Die Komponenten sind meist – wenn sie nicht sowieso eine durch 8 teilbare Anzahl von Bits belegen – links (seltener rechts) mit Null-Bits aufgefüllt. Die Adresse des ersten belegten Bytes eines Verbunds nennt man seine *Basis-Adresse*; die Adresse einer einzelnen Komponente errechnet sich als Summe der Basisadresse und der jeweiligen *Relativadresse* der Komponente. Ungepackte Verbunde lassen sich leicht manipulieren.

Ein *gepackter Verbund* belegt genau soviel Speicherplatz, wie alle einzelnen Komponenten zusammen; die Komponenten beginnen nicht unbedingt an Byte-Grenzen. Bei der Adressierung sind deshalb Byte- und Bit-Adressen zu berücksichtigen, was die Bearbeitung kompliziert werden läßt. Gepackte Verbunde verwendet man häufig, wenn die Komponenten Bits oder Nibbles sind.

Bei *Verbunden mit Varianten* (engl. variant records) sind die Typen einzelner Komponenten von sogenannten *Diskriminatoren* abhängig; die Typen der Komponenten gehen damit aus dem Speicherinhalt des Verbunds hervor. Verbunde mit Varianten können als gepackte oder ungepackte Verbunde auftreten.

17.1 Ungepackte Verbunde

Wir betrachten als Beispiel für einen ungepackten Verbund eine Karteikarte eines Unternehmens. Die einzelnen Komponenten sollen folgendermaßen beschaffen sein:

Vorname	20 Zeichen
Name	20 Zeichen
Geburtsdatum	6 Zeichen
Monatsverdienst	13 Bits
Abteilung	3 Bits

Der Monatsverdienst benötigt zwar nur 13 Bits, da wir aber einen ungepackten Verbund aufbauen wollen, belegen wir 2 volle Bytes; ebenso verwenden wir für die 3 Bits der Komponente Abteilung ein volles Byte. Eine Speicherdefinition eines uninitialisierten Verbunds obigen Typs hat damit folgende Form:

```
VORNAM:   DEFS    20          ; Vorname
NAME:     DEFS    20          ; Name
GEBURT:   DEFS    6           ; Geburtsdatum
VERDIE:   DEFS    2           ; Monatsverdienst
ABTEIL:   DEFS    1           ; Abteilung
```

Ein Beispiel für einen initialisierten Verbund obigen Typs wäre:

```
VORNAM:   DEFM    'Otto       ' ; Vorname
NAME:     DEFM    'Huber      ' ; Name
GEBURT:   DEFM    '260448'      ; Geburtsdatum
VERDIE:   DEFW    3264          ; Monatsverdienst
ABTEIL:   DEFB    3             ; Abteilung
```

Wollen wir diesen Verbund bearbeiten, so könnten wir indirekte Adressierung mittels eines der Register BC, DE, HL benutzen. Übergeben wir beispielsweise die Basisadresse im HL-Register und wollen wir dann den Monatsverdienst ins DE-Register holen, so würde die Adressierroutine lauten:

```
          LD        DE,46       ; Relativadresse des
                                ; Monatsverdiensts
          ADD       HL,DE       ; Adresse des
                                ; Monatsverdiensts
          LD        E,(HL)      ; Monatsverdienst
          INC       HL          ; ins DE-Register
          LD        D,(HL)      ; bringen
```

Der zugehörige Objekt-Code lautet:

Adresse	Objekt-Code	Marke	Anweisung	
0000	11 2E 00		LD	DE,46
0003	19		ADD	HL,DE
0004	5E		LD	E,(HL)
0005	23		INC	HL
0006	56		LD	D,(HL)

Bereits an diesem kleinen Beispiel werden einige Mängel der Adressierung von Verbunden mittels der Doppelregister sichtbar:

Der Zeiger auf den Verbund wird – wenn wir ihn nicht explizit sichern – durch die Adressierung verändert.

Zur Berechnung der Adresse einer Komponente benötigen wir arithmetische und/oder inkrementierende/dekrementierende Befehle; bei Verwendung arithmetischer Befehle wird neben dem Daten-Adreß-Register ein weiteres Doppelregister belegt.

Wird auf mehrere Komponenten zugegriffen, so hängt die Berechnung der Adresse einer Komponente von der zuvor ausgeführten Operation ab; der Programmierer verliert bei komplexeren Vorgängen rasch den Überblick. Wird die Struktur des Verbunds geändert, so muß das Programm in der Regel neu geschrieben werden.

Gemeinsame Verwendung eines Programmstücks für die Adressierung von Komponenten eines Verbunds durch mehrere Programme ist nur unter erschwerten Bedingungen möglich.

Eine Abhilfe schaffen hier die Indexregister IX und IY. Ein Indexregister zeigt (während eines komplexen Adressierungsvorgangs) stets auf eine feste Adresse (meist die Basisadresse eines Verbunds). Die Adressierung einer Komponente eines Verbunds geschieht dabei durch Angabe der Relativ-Adresse der Komponente (relativ zum Wert des Indexregisters); die Relativ-Adresse muß dem Bereich −80H bis +7FH entstammen. Obiges Beispiel würde mit Hilfe des Indexregisters IX folgendermaßen lauten, wenn IX zu Beginn auf den Anfang des Verbunds zeigt:

```
LD      E,(IX+46)
LD      D,(IX+47)
```

Das Programmstück ist entschieden kürzer als das zuvor mittels des HL-Registers programmierte; allerdings trügt hier etwas der Schein, denn der Objekt-Code des zweiten Programms belegt immerhin auch 6 Bytes, im Gegensatz zu den 8 Bytes der ersten Variante:

Adresse	Objekt-Code	Marke	Anweisung	
0000	DD 5E 2E		LD	E,(IX+46)
0003	DD 56 2F		LD	D,(IX+47)

Insbesondere wenn hintereinanderliegende Bytes eines Verbunds bearbeitet werden, kann die Adressierung durch Indexregister mehr Objekt-Code belegen als die Adressierung durch ein Doppelregister. Trotzdem ist die Adressierung durch Indexregister übersichtlicher, weniger fehleranfällig, konsequenter und gut modifizierbar; in der Adressierung mehrerer Komponenten sind niemals Abhängigkeiten der Adreßberechnung enthalten.

Die Indexregister IX und IY, die sich völlig gleich verhalten, können in fast allen Befehlen, die das HL-Register benutzen, dieses ersetzen (vergleiche hierzu den Anhang A). Für den Prozessor wird diese Substitution durch ein vorangestelltes Byte (DDH für IX, FDH für IY) gekennzeichnet; dadurch wächst natürlich der Umfang des Objekt-Codes. Bei Verwendung der Indexregister zur indirekten Adressierung von Daten kommt noch die Relativadresse hinzu, die ebenfalls ein Byte belegt. Nachfolgend eine Auswahl von Befehlen für die Indexregister:

```
LD      (IY-12),L
LD      C,(IX+74H)
BIT     5,(IX+2)
RES     7,(IY+00H)
SET     2,(IY+22)
ADC     A,(IX-3)
OR      (IY+04H)
CP      (IY-56H)
INC     (IX+5)
DEC     (IY+62H)
SUB     (IX+1)
SRA     (IX-7)
RRC     (IX+12H)
LD      IX,227FH
LD      IY,(6679H)
LD      (3167H),IX
INC     IY
DEC     IX
ADD     IX,DE
ADD     IY,IY
JP      (IY)
```

Wir studieren nun noch zwei häufig vorkommende Anwendungen von ungepackten Verbunden: Deskriptoren und Kontrollblöcke.

Deskriptoren haben wir bereits bei den Feldern kennengelernt; nur war an dieser Stelle noch nicht klar, daß es sich dabei um Verbunde handelt. Eine typische Anwendung sind Deskriptoren für Zeichenketten-Variablen in BASIC-Interpretern. Wir nehmen dazu an, daß der Variablenname aus zwei Zeichen besteht und die Länge der Zeichenketten durch ein Byte angegeben wird. Der Deskriptor sieht dann meist so aus:

```
NAME:    DEFS    2       ; Variablenname
LAENGE:  DEFS    1       ; Laenge der Zeichenkette
ADRESS:  DEFS    2       ; Adresse des Texts
```

Wir wollen nun die Länge der Zeichenkette ins B-Register bringen und die Anfangsadresse des Texts ins HL-Register, falls der Variablenname mit einem im DE-Register gegebenen Variablennamen übereinstimmt; die Basis-Adresse des Deskriptors steht dabei im IY-Register:

```
         LD      A,D          ; erstes Zeichen
         CP      (IY+00H)     ; des Namens testen
         JP      NZ,FERTIG    ; Name verschieden
         LD      A,E          ; zweites Zeichen
         CP      (IY+01H)     ; des Namens testen
         JP      NZ,FERTIG    ; Name verschieden
         LD      B,(IY+02H)   ; Laenge des Texts holen
         LD      L,(IY+03H)   ; Adresse des
         LD      H,(IY+04H)   ; Texts holen
FERTIG:  NOP                  ; gemeinsame Fortsetzungsstelle
```

Am Null-Flag erkennt man, ob der Name übereinstimmt.

Ein Kontrollblock besteht aus Informationen über den Zustand eines externen Geräts sowie über die Ansprechbarkeit spezieller Treiber. Wir betrachten als Beispiel einen Drucker-Kontrollblock mit folgender Struktur:

```
NAME:    DEFS    2       ; Geraete-Name, spezifiziert
                         ; das angeschlossene Geraet
ZLAENG:  DEFS    1       ; Zeilenlaenge
ZPOSIT:  DEFS    1       ; aktuelle Position des
                         ; Druckkopfes in der Zeile
SLAENG:  DEFS    1       ; Seitenlaenge
SPOSIT:  DEFS    1       ; aktuelle Position des
                         ; Druckkopfes auf der Seite
TREIBA:  DEFS    2       ; Adresse des Treiber-Programms
```

Zeigt IX auf den Kontrollblock, so bringt folgendes Programm die Koordinaten des Druckkopfes auf dem Papier ins DE-Register:

```
ZPOS     EQU     ZPOSIT-NAME  ; Relativ-Adresse der
                              ; Position in der Zeile
SPOS     EQU     SPOSIT-NAME  ; Relativ-Adresse der
                              ; Position auf der Seite
         LD      E,(IX+ZPOS)  ; Position in der Zeile
         LD      D,(IX+SPOS)  ; Position auf der Seite
```

Wollen wir die Struktur des Kontroll-Blocks ändern, so genügt eine Korrektur der Größen ZPOS und SPOS, um das Programm an die Änderung anzupassen.

Übungen

1. Definiere zu folgendem Datensatz einen geeigneten Verbund:

   Alter (in Jahren)
   Größe (in cm)
   Gewicht (in kg)

   Schreibe ein Programm, das feststellt, ob die Person, zu welcher der Datensatz gehört, älter als 55 Jahre, kleiner als 182 cm und mindestens 78 Kilo schwer ist. Stelle fest, wieweit das Gewicht sich vom Normalgewicht unterscheidet (Normalgewicht = Größe −100).

2. Schreibe ein Programm, das in obigem Drucker-Kontrollblock nach Ausgabe eines Zeichens die aktuelle Position korrigiert.

3. Gegeben sei ein Feld, dessen Komponenten die (X,Y-)Koordinaten von Punkten in einer Ebene darstellen (als Verbund von zwei ganzen Zahlen der Länge 1 Byte zu implementieren). Finde einen Punkt, der vom Koordinatenursprung (0,0) möglichst weit entfernt ist.

17.2 Gepackte Verbunde

In einem gepackten Verbund folgen alle Komponenten so dicht wie nur möglich aufeinander, um so wenig Speicherplatz wie möglich zu belegen. Wir realisieren unsere Karteikarte aus dem vorhergehenden Unterkapitel als gepackten Verbund:

```
VORNAM:   DEFS      20          ; Vorname
NAME:     DEFS      20          ; Name
GEBURT:   DEFS      6           ; Geburtsdatum
VERABT:   DEFS      2           ; Monatsverdienst und Abteilung
```

Aus der Struktur des Verbunds kann man nicht mehr erkennen, wo die Komponente Monatsverdienst endet und die Komponente Abteilung beginnt; dies wird erst wieder aus den Algorithmen klar. Wir präsentieren ein Programm, das die Nummer der Abteilung im B-Register und den Monatsverdienst im HL-Register erwartet, die beiden Angaben packt und sie im Verbund ablegt, dessen Basis-Adresse im IX-Register steht:

```
          RR        B           ; Bit 2 wird
          RR        B           ; zuerst
```

```
        RR      B                       ; gebraucht
        ADC     HL,HL                   ; Bit 2 von B nach HL bringen
        RL      B                       ; Bit 1 von
        ADC     HL,HL                   ; B nach HL bringen
        RL      B                       ; Bit 0 von
        ADC     HL,HL                   ; B nach HL bringen
        LD      (IX+VERABT-VORNAM),L    ; die beiden gepackten
        LD      (IX+VERABT-VORNAM+1),H  ; Komponenten abspeichern
```

Ein ganz typischer Fall liegt vor, wenn die Komponenten eines Verbunds Bits sind. Beispielsweise betrachten wir Verbunde, deren erste Komponente vom Typ Bit angibt, ob ein bestimmter Graphikpunkt gesetzt ist oder nicht, und deren zweite Komponente vom Typ Bit angibt, ob der Zustand des Graphikpunkts geändert werden darf (so etwas nennt man ein Attribut des Punkts). Die folgende Routine löscht alle Graphikpunkte eines Felds (die Anzahl der Bytes des Felds im DE-Register, IX zeigt auf den Anfang des Felds), die nicht durch das Attribut geschützt sind (die Verwendung der Indexregister ist für diese Routine nicht optimal):

```
TEST:   LD      A,D             ; restliche Anzahl von Bytes
        OR      E               ; auf Null testen
        JP      Z,FERTIG        ; ganzes Feld bearbeitet
        LD      B,4             ; Anzahl der Punkte pro Byte
LOESCH: RLC     (IX+00H)        ; Attribut holen
        P       C,GESCH         ; geschuetzter Punkt
        RES     7,(IX+00H)      ; Punkt loeschen
GESCH:  RLC     (IX+00H)        ; Punkt rotieren
        DJNZ    LOESCH          ; 4 Punkte bearbeiten
        INC     IX              ; auf naechstes Byte zeigen
        DEC     DE              ; restliche Anzahl von Bytes
                                ; berechnen
        JP      TEST            ; ganzes Feld abarbeiten
FERTIG: NOP                     ; Fortsetzungsstelle
```

Übungen

1. Vereinbare einen gepackten Verbund, der aus folgenden Komponenten besteht:

Sekunde	6 Bits
Minute	6 Bits
Stunde	5 Bits
Tag	5 Bits
Monat	4 Bits
Jahr	11 Bits
Wochentag	3 Bits

Verwende folgende Codierung für den Wochentag: 0 = Montag, 1 = Dienstag, ..., 6 = Sonntag.
Schreibe nun ein Programm, das den Inhalt des Verbunds um eine Sekunde fortschaltet (wird dabei eine Minute voll, so werden auch die Minuten aktualisiert, usw).

17.3 Verbunde mit Varianten

Bisher haben wir angenommen, daß der Typ einer Komponente eines Verbunds stets derselbe ist. Nun kann es aber auch vorkommen, daß ein Verbund Daten darstellen soll, deren Typen wechseln (zum Beispiel könnte ein Programm folgende Eingabe akzeptieren: symbolischer Name einer Variablen oder Speicheradresse derselben); auch die Zahl der Komponenten variiert unter Umständen (wenn wir zum Beispiel geometrische Figuren beschreiben wollen, so brauchen wir für ein Rechteck zwei Bestimmungsgrößen - Länge und Breite, jedoch nur eine für einen Kreis - den Radius). Man gibt deshalb in diesen Fällen zu den eigentlichen Komponenten zusätzliche Komponenten - Diskriminatoren - hinzu, die bestimmen, welche Form der Verbund nun wirklich haben soll. Das Beispiel mit den geometrischen Figuren könnte als ungepackter Verbund lauten:

```
SELEKT:   DEFS    1     ; Diskriminator, 0 steht fuer Kreis,
                        ; 1 steht fuer Rechteck
RADIUS:
LAENGE:   DEFS    2     ; Radius des Kreises
                        ; beziehungsweise Laenge des
                        ; Rechtecks
BREITE:   DEFS    2     ; Breite des Rechtecks,
                        ; unbenutzt bei Kreisen
```

Je nach Wert des Diskriminators enthält die erste Komponente den Radius eines Kreises oder die Länge eines Rechtecks; die zweite Komponente wird nur für Rechtecke benutzt. Der von einem Verbund belegte Speicherplatz ist in der Regel stets gleich groß, welche Struktur auch immer im Inneren verwendet wird. Durch die Überlagerung der beiden alternativen Komponenten Radius und Länge wird Speicherplatz eingespart; ist eine Überlagerung nicht wünschenswert, so verwende man Verbunde ohne Varianten, in denen der Diskriminator eine normale Komponente darstellt. Noch schnell zwei Beispiele für die Anwendung obiger Struktur:

```
KREIS:    DEFB    0       ; Diskriminator fuer Kreis
RADIUS:   DEFW    1200    ; Radius des Kreises
LEER:     DEFS    2       ; unbenutze Komponente

RECHTE:   DEFB    1       ; Diskriminator fuer Rechteck
LAENGE:   DEFW    940     ; Laenge des Rechtecks
BREITE:   DEFW    385     ; Breite des Rechtecks
```

Eine mögliche Aufgabe für die Bearbeitung einer solchen Struktur könnte darin bestehen, die Fläche der Figur (angenähert) auszurechnen. Wir schreiben deshalb ein Programm, das für einen Kreis dessen Radius ins BC-Register bringt und das Null-Flag setzt, für ein Rechteck dagegen Länge und Breite in die Register HL und DE bringt und das Null-Flag löscht; IY soll die Basis-Adresse des Verbunds enthalten:

```
        XOR     A           ; Testgroesse bereitstellen
        OR      (IY+00H)    ; Diskriminator testen
        JP      Z,KREIS     ; Verbund stellt Kreis dar
        LD      L,(IY+01H)  ; Laenge des
        LD      H,(IY+02H)  ; Rechtecks holen
        LD      E,(IY+03H)  ; Breite des
        LD      D,(IY+04H)  ; Rechtecks holen
        JP      FERTIG      ; Aufgabe geloest
KREIS:  LD      C,(IY+01H)  ; Radius des
        LD      B,(IY+02H)  ; Kreises holen
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Als Beispiel für Varianten mit gleicher Anzahl von Komponenten, aber differierenden Komponententypen sei folgendes genannt: Wir holen vier Eingabezeichen von der Tastatur und prüfen, ob diese eine Speicheradresse in Hex-Schreibweise darstellen. Ist dies der Fall, so konvertieren wir die vier Zeichen in die dargestellte Adresse; ansonsten legen wir die Zeichen ungeändert ab und interpretieren sie später als Name einer Variablen. Die angemessene Struktur für das Problem ist folgender Verbund mit Variante:

```
SELEKT: DEFS    1           ; Diskriminator, 0 steht fuer Adresse,
                            ; 1 steht fuer Name
ADRESS:
NAME:   DEFS    4           ; Name einer Variablen belegt
                            ; vier Bytes, Adresse einer
                            ; Variablen belegt die ersten
                            ; beiden Bytes, folgende Bytes
                            ; sind ungenutzt
```

In Assemblerschreibweise läßt sich die Struktur komplizierter Verbunde nicht sinnvoll ausdrücken; eine korrekte Interpretation der Datenstruktur ist deshalb nur aus den Algorithmen und den Kommentaren zu ersehen.

Verbunde mit Varianten kann man ebenfalls packen, um Speicherplatz zu sparen; das kann insbesondere deshalb sinnvoll sein, weil der Diskriminator normalerweise nur wenige Bits benötigt.

Übungen

1. Im D-Register und im E-Register stehe jeweils ein Zeichen. Schreibe ein Programm, das einen Verbund mit Varianten anlegt. Sind beide Zeichen ASCII-codierte Dezimalziffern, so soll die entsprechende Zahl berechnet und als Byte gespeichert werden; ansonsten sind die beiden Zeichen ungeändert abzulegen.

2. Definiere einen Verbund mit Varianten, der wahlweise ein Dreieck, ein Quadrat, ein Rechteck oder einen Kreis beschreibt.

18
Der Stapel

Der *Stapel* (engl. stack) ist eine der faszinierendsten Datenstrukturen, die ich kenne, und aus der Assemblerprogrammierung einfach nicht wegzudenken (seine wichtigste Aufgabe - die Unterstützung von Unterprogrammen - werden wir im nächsten Kapitel kennenlernen).

Unter einem Stapel kann man sich getrost einen Stapel von Paketen vorstellen. Es sind nur zwei Operationen auf dem Stapel zulässig: das Aufschichten eines weiteren Pakets auf das oberste Paket des Stapels und das Herunternehmen des obersten Pakets des Stapels. Beim Stapel des Z80 sind die Pakete stets Worte.

Der Stapel wird manchmal auch *Keller* genannt. Eine weit verbreitete Bezeichnung in der englischsprachigen Literatur ist *LIFO*; das steht für »last in, first out« und charakterisiert die Eigenschaft des Stapels, daß jeweils nur das oberste - das heißt zuletzt abgelegte - Paket zugänglich ist.

18.1 Die natürlichen Stapel-Operationen

Der vom Z80 durch spezielle Befehle unterstützte Stapel (»Hardware-Stapel« des Z80) kann in einem beliebigen Speicherbereich untergebracht sein; es ist ein hängender Stapel, das heißt, daß der Stapel von den höheren zu den niedrigeren Speicheradressen wächst (das oberste Element des Stapels hat damit immer die kleinste Adresse von allen Stapelelementen). Zur Unterstützung des Stapels besitzt der Z80 ein spezielles Register, den *Stapel-Zeiger* **SP** (engl. stack pointer); dieser zeigt stets auf das oberste Element des Stapels, genauer auf dessen LSB, und damit auf das Byte mit der kleinsten Adresse, das vom Stapel belegt wird.

Bevor mit dem Stapel gearbeitet werden kann, muß der Stapel-Zeiger einen definierten und sinnvollen Wert erhalten. Normalerweise initialisiert man den Stapel-Zeiger deshalb so, daß er einen leeren Stapel markiert, das ist ein Stapel, der noch kein Element enthält. Man reserviert für den Stapel einen bestimmten Bereich des Speichers, zum Beispiel ab der Adresse

ANFANG bis einschließlich der Adresse ENDE (ENDE soll die höhere Adresse sein). Bei leerem Stapel zeigt der Stapel-Zeiger dann direkt hinter den reservierten Speicherbereich, also auf die Adresse ENDE+1.

Das Initialisieren des Stapel-Zeigers geschieht mit einem LD-Befehl. Der Stapel-Zeiger kann dabei mit einem konstanten Wert, dem Inhalt einer Variablen vom Typ Wort, dem Inhalt des HL-Registers oder dem Inhalt eines Indexregisters geladen werden. Beispielsweise:

```
ANFANG   EQU      7800H      ; Adresse des obersten
                              ; nutzbaren Bytes des Stapels ENDE
         EQU      7FFFH      ; Adresse des untersten
                              ; nutzbaren Bytes des Stapels

; Programmbereich

         LD       SP,ENDE+1  ; leeren Stapel aufsetzen

; Datenbereich

         ORG      ANFANG
STAPEL:  DEFS     ENDE-ANFANG+1      ; Speicherplatz fuer Stapel
                                     ; reservieren
```

Beachte, daß »oben« und »unten« sich immer auf den Stapel bezieht, nicht jedoch auf den Speicher (das oberste Element hat die kleinste Adresse).

Mittels des Befehls PUSH (push) bringen wir den Wert eines Doppelregisters (AF, BC, DE, HL) oder Indexregisters (IX, IY) auf den Stapel, ohne das gesicherte Register zu verändern:

```
         PUSH     AF         ; AF-Register sichern
         PUSH     HL         ; HL-Register sichern
         PUSH     IY         ; IY-Register sichern
```

Die Abbildung, Bild 18.1., erläutert die Funktionsweise des PUSH-Befehls.

Die Adressierung des Stapels geschieht indirekt. Der Befehl PUSH bezieht sich implizit auf den Stapel-Zeiger SP (das SP-Register braucht in der Assemblerschreibweise also nicht explizit angegeben zu werden). Die Wirkung des Befehls PUSH HL zum Beispiel kann formal folgendermaßen beschrieben werden:

SP <– <SP> – 1
(<SP>) <– <H>
SP <– <SP> – 1
(<SP>) <– <L>

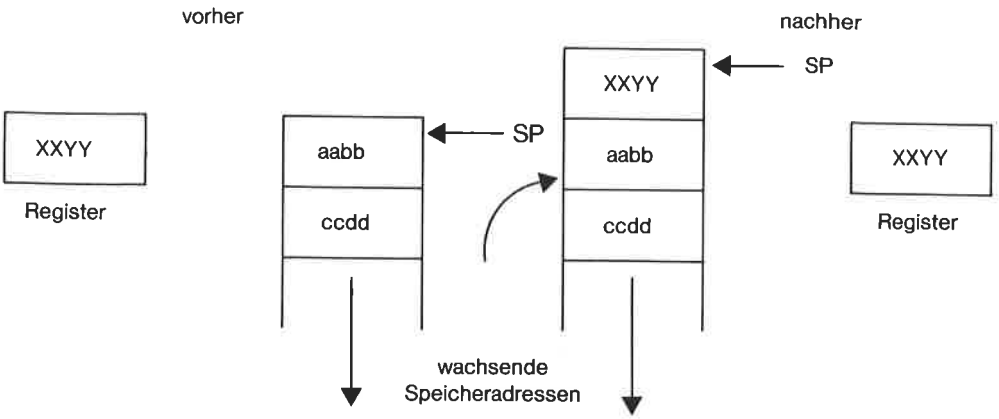


Bild 18.1. *Wirkung des PUSH-Befehls*

Nach Ausführung des PUSH-Befehls weist der Stapel-Zeiger wieder auf das LSB des obersten (das heißt zuletzt abgelegten) Stapel-Elements. Das LSB kommt immer in die Speicherzelle mit der niedrigeren Adresse, so wie es auch beim LD-Befehl gehandhabt wird.

Ein PUSH-Befehl kann natürlich nur ausgeführt werden, wenn auf dem Stapel noch Platz für ein Wort ist, das heißt, wenn der Stapel-Zeiger vor dem Ausführen der Operation einen Wert nicht kleiner als ANFANG+2 besitzt; sonst würde ein sogenannter *Stapel-Ueberlauf* die Folge sein. Manchmal weiß man (durch Studieren der Algorithmen, die den Stapel verwenden), daß diese Bedingung erfüllt ist; dann kann man die Operation ohne weiteres ausführen. Ansonsten muß eine Zulässigkeitsprüfung erfolgen. Der Z80 besitzt deshalb einen 16-Bit-SBC-Befehl, in dem das Register SP vorkommt:

```
LD      HL,ANFANG+1  ; Testgroesse laden
OR      A            ; Uebertrag-Flag loeschen
SBC     HL,SP        ; auf Stapel-Ueberlauf testen
JP      NC,FEHLER    ; Fehler: Stapel-Ueberlauf
```

Kommt es durch Unterlassung einer Zuverlässigkeitsprüfung (einer der häufigsten Programmierfehler!) zu einem Stapel-Überlauf, so werden einige Bytes unterhalb des Stapels zerstört; enthalten sie Code oder Daten, so resultieren meist bösartige Programmfehler.

Das oberste Element des Stapels wird durch den Befehl POP (pop) vom Stapel entfernt und in ein Doppelregister oder Indexregister gebracht; der alte Inhalt des Registers wird dabei zerstört (Bild 18.2.).

Einige Beispiele:

```
POP     BC           ; BC-Register restaurieren
POP     IX           ; IX-Register restaurieren
```

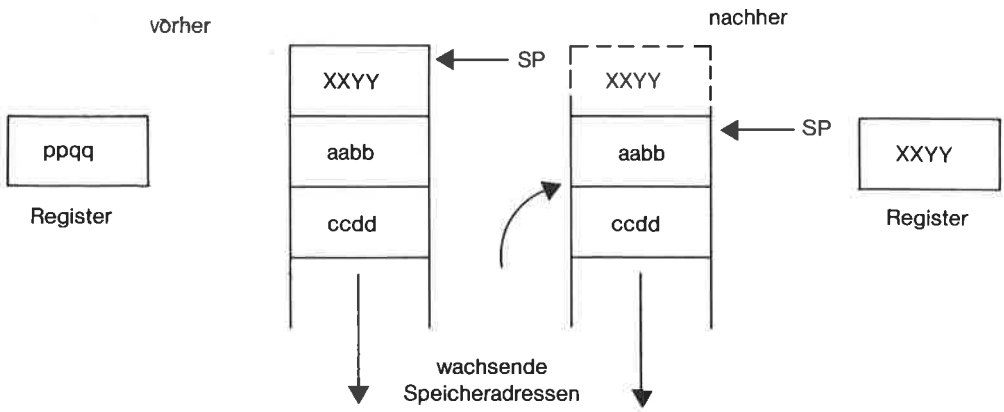


Bild 18.2. *Wirkung des POP-Befehls*

Die Wirkung des Befehls POP BC kann formal folgendermaßen beschrieben werden:

C <– <(<SP>)>
SP <– <SP> + 1
B <– <(<SP>)>
SP <– <SP> + 1

Wieder zeigt der Stapel-Zeiger nach Abschluß der Operation auf das LSB des obersten Stapel-Elements. Das geholte Stapel-Element verschwindet nur logisch vom Stapel, während es im Speicher weiterhin stehenbleibt, bis es durch einen nachfolgenden PUSH-Befehl überschrieben wird; bis dahin kann es wie der Inhalt einer Variablen benutzt werden (aber bitte mit Vorsicht!). Eine Besonderheit des Befehls POP AF ist, daß durch ihn die Flags verändert werden; sie erhalten die Werte der entsprechenden Bits aus dem LSB des obersten Stapel-Elements. Dies ist der einzige Befehl, mit dem man allen Flags bestimmte Werte zuweisen kann.

Wenn wir mehr Elemente vom Stapel nehmen wollen als dort vorhanden sind, so resultiert ein *Stapel-Unterlauf*; der Stapel-Zeiger verläßt wie beim Stapel-Überlauf den zulässigen Bereich. Obwohl durch einen Stapel-Unterlauf ein ungültiges Stapel-Element geholt wird, führt er nicht direkt zur Zerstörung von Code oder Daten; allerdings hat ein nachfolgender PUSH-Befehl eben diese Wirkung. Bei korrekt programmierten Algorithmen kann es niemals zu einem Stapel-Unterlauf kommen; es kann jedoch aus Gründen erhöhter Programmsicherheit ratsam sein, vor Durchführung eines POP-Befehls auf Stapel-Unterlauf (<SP> > ENDE–1) zu testen:

```
LD      HL,ENDE     ; Testgroesse laden
SCF                 ; Uebertrag-Flag setzen
SBC     HL,SP       ; auf Stapel-Unterlauf testen
JP      C,FEHLER    ; Fehler: Stapel-Unterlauf
```

Wir können auf dem Stapel nun Registerinhalte aufbewahren, die durch eine nachfolgende Berechnung sonst zerstört würden, und den ursprünglichen Inhalt nach der Operation wiederherstellen. Steht beispielsweise im DE-Register die Anfangsadresse eines Wort-Felds, im HL-Register der Index eines Feldelements, und wollen wir den Inhalt dieses Feldelements ins BC-Register holen, so können wir - bei korrekt aufgesetztem Stapel - den Inhalt des HL-Registers retten (wir lassen den Test auf Stapel-Überlauf weg):

```
PUSH    HL      ; HL-Register retten
ADD     HL,HL   ; Index zu Relativadresse machen
ADD     HL,DE   ; Adresse des Elements berechnen
LD      C,(HL)  ; Inhalt des
INC     HL      ; Feldelements
LD      B,(HL)  ; holen
POP     HL      ; HL-Register restaurieren
```

Wir können auch mehrere Register gleichzeitig auf den Stapel retten; dabei müssen wir allerdings beim Restaurieren die Reihenfolge der Register vertauschen. Wenn wir beispielsweise nach dem Kopieren eines Byte-Felds mittels LDIR die alten Registerinhalte wieder benötigen, könnte dies so aussehen:

```
PUSH    BC      ; BC retten
PUSH    DE      ; DE retten
PUSH    HL      ; HL retten
LDIR            ; Kopiervorgang ausfuehren
POP     HL      ; Register wieder
POP     DE      ; restaurieren, beachte die
POP     BC      ; Reihenfolge!
```

Nicht mehr benötigte Stapel-Elemente sollten vom Stapel entfernt werden. Dies kann durch einen POP-Befehl geschehen, wenn der Inhalt des betroffenen Registers unwichtig ist; ansonsten inkrementiert man zweimal den Stapel-Zeiger:

```
INC     SP      ; nicht mehr benoetigtes
INC     SP      ; Stapel-Element entfernen
```

Manchmal ist es recht lästig, daß es nur einen Stapel-Zeiger gibt. Wir können uns aber trotzdem mehrere Stapel schaffen, indem wir mehrere Speicherbereiche als Stapel aufsetzen und den Stapel-Zeiger rechtzeitig mit der richtigen Adresse laden. Dazu müssen wir den alten Wert des Stapel-Zeigers in eine Variable retten. Wir betrachten ein System mit zwei Stapeln. Nehmen wir an, wir hätten zwei Variablen für die aktuellen Werte der beiden Stapel-Zeiger definiert und gerade noch auf dem ersten Stapel gearbeitet; nun wollen wir auf die Bearbeitung des zweiten Stapels umschalten:

```
; Programmbereich

            LD          (SP1),SP        ; Stapel-Zeiger von
                                        ; Stapel 1 sichern
            LD          SP,(SP2)        ; alten Stapel-Zeiger von
                                        ; Stapel 2 laden

; Datenbereich

SP1:        DEFW        ENDE1           ; Stapel-Zeiger fuer Stapel 1
SP2:        DEFW        ENDE2           ; Stapel-Zeiger fuer Stapel 2
STAP1:      DEFS        128             ; Stapel 1 (64 Elemente)
ENDE1:
STAP2:      DEFS        32              ; Stapel 2 (16 Elemente)
ENDE2:
```

Das Retten des Stapel-Zeigers kann auch sinnvoll sein, bevor in unübersichtlicher Reihenfolge Daten auf den Stapel gebracht werden; man kann hinterher den Stapel-Zeiger dann wieder auf einen definierten Wert setzen. Der Stapel-Zeiger kann auch in eines der Register HL, IX oder IY gerettet werden, zum Beispiel

```
            LD          HL,0            ; Vorbereitung fuer Retten von SP
            ADD         HL,SP           ; SP in HL-Register retten
```

oder

```
            LD          IX,0            ; Vorbereitung fuer Retten von SP
            ADD         IX,SP           ; SP in IX-Register retten
```

Eine zweite wichtige Aufgabe des Stapels ist es, die Vertauschung von Registerinhalten zu unterstützen. Wir haben zwar bereits die Ringtausch-Methode kennengelernt; diese benötigt aber immer ein Hilfsregister (HL und DE können jedoch direkt durch EX DE,HL vertauscht werden). Außerdem funktioniert diese Methode nur bei einfachen und Doppelregistern, nicht aber bei den Indexregistern. Wollen wir den Inhalt eines Doppelregisters oder Indexregisters in ein anderes übertragen, so bringen wir den Wert erst auf den Stapel und holen ihn von dort wieder ab. Mit folgendem Programmstück kopieren wir den Inhalt des BC-Registers ins IY-Register:

```
            PUSH        BC              ; Wert auf den Stapel bringen
            POP         IY              ; Wert vom Stapel abholen
```

Auf die gleiche Art können wir alle Flags gleichzeitig in einem Register sichern, wobei die andere Hälfte des betroffenen Doppelregisters aber zerstört wird:

```
PUSH    AF      ; Flags auf den Stapel werfen
POP     DE      ; Flags ins E-Register bringen,
                ; D-Register wird zerstört
```

Manchmal kommt es vor, daß wir den Inhalt eines Registers auf dem Stapel sichern wollen, das oberste Element des Stapels aber gleich darauf in diesem Register benötigt wird. Hier helfen uns einige Austauschbefehle weiter:

```
EX      (SP),HL ; Inhalt des HL-Registers
                ; sichern, gleichzeitig oberstes
                ; Stapel-Element holen
```

Denselben Befehl gibt es auch für die Indexregister. Wir beschreiben obigen Befehl formal:

HL & (<SP>) <– <(<SP>)> & <HL>

Die Funktion des EX-Befehls verdeutlicht folgende Abbildung:

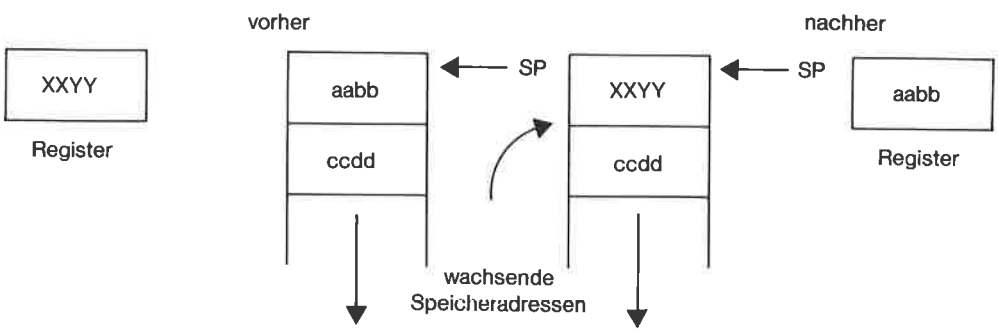


Bild 18.3. *Wirkung des EX-Befehls*

Wir können mit dem EX-Befehl auch die obersten beiden Stapel-Elemente vertauschen, wenn wir das HL-Register dafür frei haben:

```
POP     HL      ; oberstes Stapel-Element holen
EX      (SP),HL ; beide Elemente tauschen
PUSH    HL      ; zweitoberstes Stapel-Element
                ; als oberstes ablegen
```

Eine schöne Methode, um die Inhalte zweier Register zu tauschen, ohne den Stapel zu verändern, besteht in der Verwendung von drei Austausch-Befehlen. Wir vertauschen beispielsweise die Register IX und IY:

```
EX      (SP),IX     ; oberstes Stapel-Element holen,
                    ; statt dessen Inhalt des
                    ; IX-Registers ablegen
EX      (SP),IY     ; alten Wert des IX-Registers
                    ; ins IY-Register bringen,
                    ; alten Wert des IY-Registers
                    ; auf den Stapel werfen
EX      (SP),IX     ; alten Wert des IY-Registers
                    ; ins IX-Register bringen,
                    ; alten Wert des obersten
                    ; Stapel-Elements wiederherstellen
```

Für diese hübsche Tauschaktion braucht der Stapel auch gar nicht richtig aufgesetzt zu sein; es genügt, wenn der Stapel-Zeiger auf ein Wort im RAM (random access memory = beschreibbarer Speicher) weist.

Übungen

1. Definiere einen Stapel der Länge 512 Bytes, der ab der Adresse FFFFH nach unten hängt. Bringe dann die Inhalte der Register BC, DE, IX und IY auf den Stapel. Überschreibe die Register und lies dann die alten Werte vom Stapel wieder ein.

2. Tausche die Inhalte des DE-Registers und des IY-Registers.

3. Bringe in dem Beispiel mit den beiden Stapeln die zwei obersten Elemente des zweiten Stapels auf den ersten Stapel.

4. Bilde im DE-Register die Summe des HL-Registers und BC-Registers, im BC-Register die Summe des HL-Registers und DE-Registers und im HL-Register die Summe des BC-Registers und DE-Registers (für die Summation jeweils die alten Werte verwenden!). Lade die alten Werte wieder, wenn eine der Summen einen Überlauf erzeugt. Lasse den Stapel im gleichen Zustand zurück wie vor der Operation.

18.2 Byte-Operationen auf dem Stapel

Manchmal wollen wir nicht Wörter, sondern Bytes sichern. Die einfachste Möglichkeit, dies zu tun, besteht darin, das Doppelregister, in dem das Byte steht, als Ganzes zu sichern. Nachteilig ist dabei, daß beim Restaurieren des Bytes die andere Hälfte des Registers zerstört wird; außerdem verschenken wir die Hälfte des belegten Speicherplatzes.

Wir studieren nun einige Möglichkeiten, einzelne Bytes auf dem Stapel abzulegen. Steht das Byte, das wir sichern wollen, im höherwertigen Anteil eines Doppelregisters, so können wir dieses Doppelregister sichern und anschließend den Stapel-Zeiger wieder inkrementieren:

```
PUSH    AF      ; Inhalt des A-Registers
INC     SP      ; auf den Stapel bringen
```

Steht das Byte dagegen im niederwertigen Anteil eines Doppelregisters, so muß es zuvor in den höherwertigen Anteil eines Doppelregisters gebracht werden:

```
LD      B,C     ; Inhalt des C-Registers
PUSH    BC      ; auf den
INC     SP      ; Stapel bringen
```

Hat auf dem Stapel nur noch ein einziges Byte Platz, so müssen wir ein weiteres Doppelregister zu Hilfe nehmen, in welches das oberste Byte des Stapels zuerst gerettet wird:

```
DEC     SP      ; auf oberstes nutzbares Byte
                ; des Stapels zeigen
POP     HL      ; frueheres oberstes Byte des
                ; Stapels ins H-Register holen
LD      L,A     ; Inhalt des A-Registers
PUSH    HL      ; auf den Stapel bringen
```

Natürlich müssen wir beim Abholen der Bytes wieder wissen, wann ein Byte und wann ein Wort auf dem Stapel liegt; dies ist Sache der Programmlogik. Beim Zurückholen des Bytes wird auf jedem Fall ein Doppelregister benötigt, dessen eine Hälfte damit unnötigerweise zerstört wird; das ist leider nicht zu vermeiden. Durch die Reihenfolge der verwendeten Befehle können wir steuern, ob das Byte in den niederwertigen oder höherwertigen Anteil des Doppelregisters kommt:

```
POP     DE      ; Byte vom Stapel ins
                ; E-Register holen
DEC     SP      ; auf oberstes Stapel-Element
                ; zeigen
```

beziehungsweise

```
DEC     SP      ; Stapel um ein Byte vergroessern
POP     DE      ; Byte vom Stapel ins
                ; D-Register holen
```

Wenn ein Byte in den höherwertigen Anteil des Doppelregisters kommen soll und auf dem Stapel noch ein Wort Platz hat, so können wir durch einen Trick den niederwertigen Anteil des Doppelregisters restaurieren:

```
LD      H,L     ; LSB des Doppelregisters
                ; in MSB umladen
```

```
PUSH    HL      ; zu restaurierendes Byte sichern
INC     SP      ; und zum LSB des obersten
                ; Elements des Stapels machen
POP     HL      ; zu restaurierendes Byte
                ; zusammen mit gewuenschtem
                ; Byte vom Stapel holen
```

Mit den angegebenen Techniken können wir nun Strukturen beliebiger Länge auf den Stapel bringen (durch Zusammensetzen aus Wort- und Byte-Sicherungsoperationen). Wenn die Strukturen in der Länge variieren, so ist es meist günstiger, den Stapel mit Hilfe einer Liste darzustellen (siehe Kapitel »Verzeigerte Strukturen«).

Übungen

1. Bringe ein Byte vom Stapel ins A-Register, ohne die Flags zu zerstören.

2. Bringe den Wert des D-Registers auf den Stapel.

3. Bringe den Wert des E-Registers auf den Stapel.

4. Hole ein Byte vom Stapel ins C-Register.

5. Hole ein Byte vom Stapel ins H-Register.

18.3 Adressierung des Stapels über andere Register

Bisher haben wir auf dem Stapel stets mit Hilfe des Stapel-Zeigers SP adressiert. Da der Stapel in einem beliebigen Speicherbereich untergebracht werden kann, können wir aber auch die schon gelernten Methoden der Datenadressierung anwenden, wenn wir uns die jeweilige Basis-Adresse der verwendeten Struktur beschaffen. Haben wir zum Beispiel die Elemente eines Felds auf den Stapel gebracht (die mit dem höchsten Index zuerst), so können wir den Wert des Stapel-Zeigers als Basis-Adresse benutzen. Nehmen wir an, daß es sich um ein Byte-Feld handelt, und daß wir den Index eines Elements im HL-Register stehen haben. Dann erfolgt ein Zugriff auf das Element mittels

```
ADD     HL,SP   ; Adresse des Feldelements
                ; berechnen
LD      A,(HL)  ; Feldelement holen
```

Wir können den Stapel (oder ein obenauf liegendes Stück davon) auch als Verbund interpretieren. Haben wir beispielsweise (in dieser Reihenfolge) die X-, Y- und Z-Koordinate eines Raum-

punkts (als Worte) auf den Stapel gebracht, und wollen wir diese nun ins BC-, DE- und HL-Register bringen, so tun wir dies durch

```
LD      IX,0        ; Vorbereitung zum Umspeichern
                    ; des Stapel-Zeigers
ADD     IX,SP       ; Stapel-Zeiger ins IX-Register
                    ; bringen
LD      C,(IX+4)    ; X-Koordinate
LD      B,(IX+5)    ; holen
LD      E,(IX+2)    ; Y-Koordinate
LD      D,(IX+3)    ; holen
LD      L,(IX+0)    ; Z-Koordinate
LD      H,(IX+1)    ; holen
```

Wenn wir große Datenstrukturen auf den Stapel bringen, so tun wir dies (wenn wir es überhaupt tun!) nicht durch viele PUSH-Befehle, sondern durch direktes Kopieren und anschließendes Laden des Stapel-Zeigers. Im folgenden Beispiel wird ein Feld auf den Stapel gebracht, auf dessen letztes Byte das DE-Register weist und dessen Länge (in Bytes) im BC-Register steht:

```
LD      HL,-1       ; Zeiger auf erstes freies Byte
ADD     HL,SP       ; des Stapels berechnen
EX      DE,HL       ; Zeiger tauschen
LDDR                ; Feld auf den Stapel werfen
EX      DE,HL       ; Zeiger tauschen
LD      SP,HL       ; Stapel-Zeiger auf das erste
INC     SP          ; Element des Felds richten
```

Übungen

1. Auf dem Stapel liegt eine Zeichenkette mit Längenangabe (so wie sie auch im normalen Datenspeicher stehen würde). Nimm diese Zeichenkette vom Stapel und lege sie ab der Adresse ab, die im HL-Register steht.

2. Das IX-Register zeigt auf einen Verbund folgender Struktur:

Bezeichnung	24 Bytes
Menge	1 Wort
Rabatt	1 Byte
Preis	1 Wort

   Bringe den Verbund auf den Stapel.

3. Auf dem Stapel liegt eine Datenstruktur mit 62 Bytes Länge, die nicht mehr benötigt wird. Korrigiere den Stapel-Zeiger entsprechend.

19
Unterprogramme

Wir haben bereits viele Programmstücke kennengelernt, die eine bestimmte, fest umrissene Aufgabe lösen (zum Beispiel das Umwandeln eines Bytes in zwei Nibbles); im Prinzip konnten wir diese Programmstücke als komplette Programme ansehen. Allerdings stellen diese Aufgaben im Rahmen eines realistischen Problems lediglich (in sich abgeschlossene) Teilaufgaben dar. Einige der betrachteten Programmstücke kamen häufiger in verschiedenen Zusammenhängen vor (zum Beispiel Multiplikationsroutinen, Überlaufprüfungen, Umwandlungen von Ziffern und Zahlen); die Programmstücke waren dabei fest in einen größeren Komplex von Befehlen eingebaut.

Es ist deshalb sinnvoll, diese immer wieder benötigten Programmstücke als eigenständige Programme – sogenannte *abgeschlossene Unterprogramme* (engl. subroutines) – zu entwickeln, zu testen und anzuwenden. Aus dem praktischen Umgang mit großen Programmsystemen weiß man, daß auf diese Art die Anzahl der Programmierfehler gesenkt werden kann. Ein großes Programm ohne Unterprogramme zu entwickeln ist nicht nur dumm, sondern meist wegen der Problemkomplexität gar nicht möglich.

Unterprogramme haben folgende Vorteile:
- sauberer Programmierstil,
- große Programme werden übersichtlich durch Zerlegung in viele kleine überschaubare Unterprogramme (Top-Down-Design),
- Unterprogramme können getrennt entwickelt und getestet werden (Modularisierung),
- ein getestetes Unterprogramm kann in vielen verschiedenen Programmen verwendet werden (Verwendung von Bibliotheken),
- Unterprogramme lassen sich gut dokumentieren,
- Unterprogramme, die mehrfach auftretende Programmstücke ersetzen, brauchen weniger Speicherplatz als die Programmstücke.

Wie wir nachher noch sehen werden, haben Unterprogramme aber auch spezifische Nachteile:
- Für die Organisation des Unterprogramms (Aufruf, Parameterübergabe, Rückkehr ins Hauptprogramm) wird zusätzliche Rechenzeit benötigt,
- Unterprogramme können nur mit Hilfe des Stapels realisiert werden; dieser kann dann für andere Zwecke nicht mit der gewohnten Freizügigkeit benutzt werden.

19.1 Aufruf und Verlassen von Unterprogrammen

Wir haben im Kapitel »Bit-Manipulationen« als Übung das Problem gelöst, ein Byte in zwei Hex-Ziffern zu zerlegen und deren ASCII-Codierung zu berechnen. Als Teilproblem trat dabei die Umrechnung einer Hex-Ziffer in ihre ASCII-Codierung auf. Wir wollen das zugehörige Programmstück zunächst noch einmal aufschreiben:

```
         CP      10              ; A-Register auf
                                 ; Dezimalziffer testen
         JP      C,DEZIMA        ; Dezimalziffer im A-Register
         ADD     A,'A'-0AH-'0'   ; Korrektur fuer Buchstaben
DEZIMA:  ADD     A,'0'           ; ASCII-Darstellung berechnen
```

Um aus dem Programmstück ein Unterprogramm zu machen, müssen wir zwei Dinge tun:

1. Denjenigen Befehl des Programmstücks, der als erster ausgeführt werden soll, müssen wir mit einer Marke versehen; nur so weiß das aufrufende Programm, welches Unterprogramm gemeint ist.
2. Nach demjenigen Befehl des Programmstücks, der als letzter ausgeführt wird, muß der Befehl **RET** (return) stehen.

Die Wirkung des RET-Befehls entspricht dem (beim Z80 nicht explizit vorhandenen) Befehl POP PC; es wird also ein Wort vom Stapel genommen und als Code-Adresse, an der fortgefahren werden soll, verwendet. Diese Adresse - die *Rückkehr-Adresse* - muß das aufrufende Programm vor der Aktivierung des Unterprogramms auf den Stapel bringen (wie das geschieht, sehen wir gleich). Unser Unterprogramm lautet nun:

```
HEXASC:  CP      10              ; A-Register auf
                                 ; Dezimalziffer testen
         JP      C,DEZIMA        ; Dezimalziffer im A-Register
         ADD     A,'A'-0AH-'0'   ; Korrektur fuer Buchstaben
DEZIMA:  ADD     A,'0'           ; ASCII-Darstellung berechnen
         RET                     ; Ruecksprung ins Hauptprogramm
```

Nun wollen wir studieren, wie das Unterprogramm aufgerufen wird:

```
        LD      B,A         ; Byte sichern
        AND     00001111B   ; niederwertigen Nibble isolieren
        CALL    HEXASC      ; Aufruf des Unterprogramms:
                            ; Nibble in ASCII codieren
        LD      C,A         ; ASCII-Code des niederwertigen
                            ; Nibbles ablegen
        LD      A,B         ; Byte wieder herstellen
        SRL     A           ; hoeher-
        SRL     A           ; wertigen
        SRL     A           ; Nibble
        SRL     A           ; isolieren
        CALL    HEXASC      ; Aufruf des Unterprogramms:
                            ; Nibble in ASCII codieren
        LD      B,A         ; ASCII-Code des hoeherwertigen
                            ; Nibbles ablegen
```

Unterprogramm und aufrufendes Programm zusammen besitzen folgenden Objekt-Code:

Adresse	Objekt-Code	Marke	Anweisung	
0000	FE 0A	HEXASC: CP	10	
0002	DA 07 00		JP	C,DEZIMA
0005	C6 07		ADD	A,'A'–0AH–'0'
0007	C6 30	DEZIMA: ADD	A,'0'	
0009	C9		RET	
000A	47		LD	B,A
000B	E6 0F		AND	00001111B
000D	CD 00 00		CALL	HEXASC
0010	4F		LD	C,A
0011	78		LD	A,B
0012	CB 3F		SRL	A
0014	CB 3F		SRL	A
0016	CB 3F		SRL	A
0018	CB 3F		SRL	A
001A	CD 00 00		CALL	HEXASC
001D	47		LD	B,A

Unser Unterprogramm wird hierbei zweimal vom Hauptprogramm aufgerufen. Die Wirkung des Befehls CALL (call) ist folgende: Zuerst wird die Adresse des auf den CALL-Befehl folgenden Befehls berechnet und auf den Stapel gebracht; dann wird die im CALL-Befehl angegebene Adresse angesprungen. Für den Rücksprung sorgt ein entsprechender RET-Befehl im Unterprogramm.

Im Flußdiagramm ausgedrückt sähe unser Programm folgendermaßen aus:

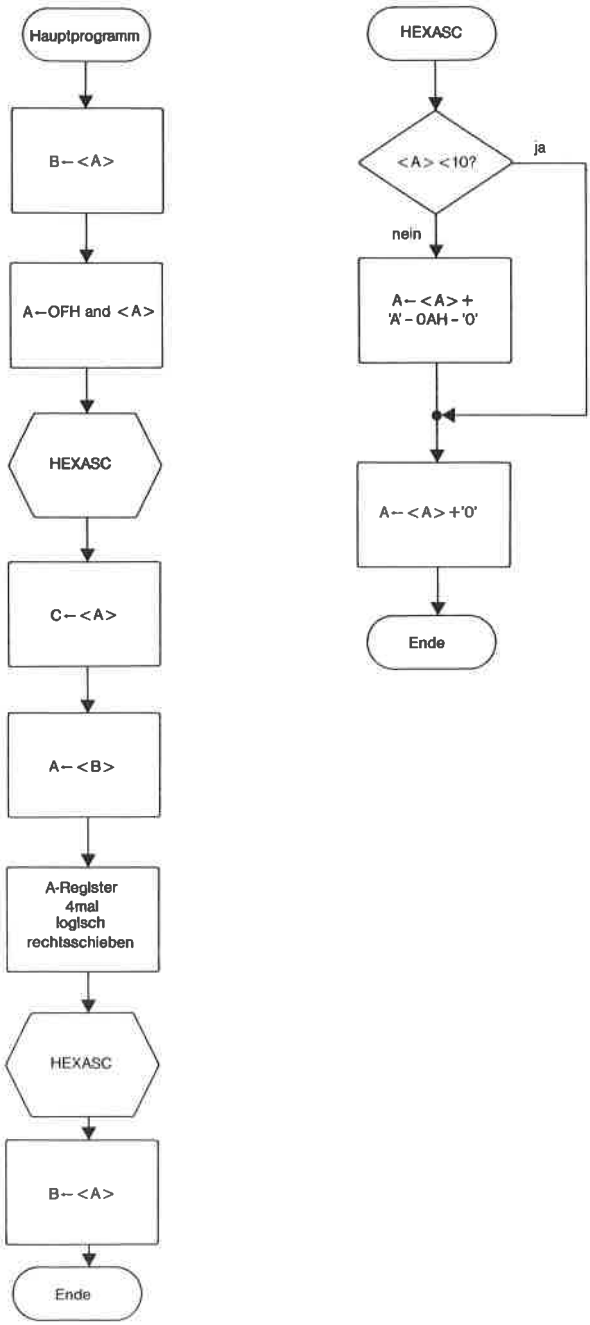


Bild 19.1. *Flußdiagramm: Unterprogramm und Unterprogramm-Aufrufe*

Schon an diesem kleinen Beispiel sieht man einige typische Phänomene der Unterprogramm-Technik:

- Für den korrekten Rücksprung aus einem Unterprogramm ist es absolut unerläßlich, daß der RET-Befehl als oberstes Stapel-Element die Rückkehradresse vorfindet. Bei der Arbeit mit dem Stapel ist dies stets zu berücksichtigen. Es ist einer der verhängnisvollsten Programmierfehler, wenn die Rückkehradresse vom Stapel entfernt wurde oder über der Rückkehradresse irrtümlich noch andere Stapelelemente liegen, wenn der RET-Befehl ausgeführt wird.
- Die Problemkomplexität wird durch den Einsatz von Unterprogrammen reduziert und die Übersichtlichkeit des gesamten Programms gesteigert.
- Das Ersetzen von gleichen Programmstücken durch ein Unterprogramm spart Speicherplatz. Je länger das Unterprogramm ist, und je häufiger es vorkommt, desto mehr Speicherplatz wird eingespart (hier sind es nur 2 Bytes: Das Programmstück hätte 9 Bytes belegt, und wäre zweimal vorgekommen; es wären also 18 Bytes notwendig gewesen. Das Unterprogramm belegt 10 Bytes, jeder CALL-Befehl 3 Bytes; dies ergibt zusammen 16 Bytes).

Auch unser Hauptprogramm würde in einem komplexen Programmsystem nur eine kleine Teilaufgabe erledigen; wir könnten es genauso zu einem Unterprogramm umgestalten, wie wir es mit HEXASC gemacht haben. Unterprogramme können also geschachtelt werden (so tief, wie es der Stapel erlaubt).

Neben dem unbedingten RET-Befehl gibt es auch eine Reihe von bedingten RET-Befehlen; die Bedingungen stimmen mit denen der absoluten bedingten Sprünge überein. Ein bedingter RET-Befehl führt nur dann zum Rücksprung, wenn die Bedingung erfüllt ist; bei nicht erfüllter Bedingung hat ein bedingter RET-Befehl keine Wirkung. Mittels eines bedingten RET-Befehls können wir das Unterprogramm HEXASC etwas optimieren:

```
HEXASC:   ADD     A,'0'          ; Dezimalziffer codieren
          CP      '0'+10         ; testen, ob es eine
                                 ; Dezimalziffer war
          RET     C              ; es war eine Dezimalziffer,
                                 ; Codierung durchgefuehrt,
                                 ; Ruecksprung
          ADD     A,'A'-0AH-'0'  ; Korrektur fuer Buchstaben
          RET                    ; Ruecksprung
```

Nun lautet das Objekt-Programm:

Adresse	Objekt-Code	Marke	Anweisung	
0000	C6 30	HEXASC:	ADD	A,'0'
0002	FE 3A		CP	'0'+10
0004	D8		RET	C
0005	C6 07		ADD	A,'A'-0AH-'0'
0007	C9		RET	

Auch ein Unterprogramm-Aufruf kann von einer Bedingung abhängig gemacht werden; wieder sind alle acht Bedingungen möglich, die wir von den absoluten bedingten Sprüngen her kennen. Betrachten wir dazu folgendes Problem: Es soll die Summe zweier ganzer 16-Bit-Zahlen in 2-Komplement-Darstellung gebildet werden. Falls das Resultat aber negativ ist, soll es zusätzlich in seinen absoluten Betrag verwandelt werden. Wir schreiben für die Negation einer 16-Bit-Zahl im HL-Register ein Unterprogramm, das wir nur dann aufrufen, wenn das Ergebnis der Addition negativ ist:

```
NEGIER:  LD      A,L        ; zuerst
         CPL                 ; das
         LD      L,A        ; 1-Komplement
         LD      A,H        ; der
         CPL                 ; Zahl
         LD      H,A        ; bilden
         INC     HL         ; ins 2-Komplement umrechnen
         RET                 ; Ruecksprung
```

Das Hauptprogramm lautet damit einfach:

```
         ADD     HL,DE      ; Summe bilden
         CALL    M,NEGIER   ; Betrag bilden
```

Die Reihenfolge von Hauptprogramm und Unterprogrammen im Speicher ist beliebig; man muß jedoch darauf achten, daß am Ende des Hauptprogramms ein Sprung ins Betriebssystem, in einen Debugger oder in einen Sprachinterpreter (welche Programmierumgebung man wählt, ist dem Programm gleich) steht, damit nicht nach den Befehlen des Hauptprogramms versehentlich Befehle eines Unterprogramms oder als Befehle interpretierte Daten ausgeführt werden.

Der Z80 besitzt noch einen Satz von acht speziellen Unterprogramm-Aufrufen, die für die Unterbrechungsbehandlung (siehe Kapitel »Unterbrechungen«) gedacht sind, aber auch in anderem Zusammenhang mit Vorteil verwendet werden können. Dies sind die Befehle RST (restart), die als Argument eine der acht speziellen Adressen 0000H, 0008H, 0010H, 0018H, 0020H, 0028H, 0030H, 0038H besitzen, also zum Beispiel

```
         RST     0018H      ; entspricht CALL 0018H
```

Die RST-Befehle wirken genauso wie CALL-Befehle; ihr Objekt-Code belegt aber nur ein Byte, und sie werden auch wesentlich schneller als ein CALL-Befehl durchgeführt. Sie eignen sich damit zum Aufruf von sehr häufig benötigten Standard-Unterprogrammen des Betriebssystems. Eine sehr interessante Anwendungsmöglichkeit für RST-Befehle ist das Setzen von Haltepunkten in Programmen durch einen Debugger. Dabei wird an der Stelle, an der das Programm angehalten werden soll, damit der Debugger die Kontrolle wieder erlangt, ein RST-Befehl eingesetzt, der einen Rücksprung in den Debugger erzwingt; da der RST-Befehl nur ein

Byte belegt, ist garantiert, daß das Anhalten wirklich nur dann erfolgt, wenn der Programmbefehl mit der gewünschten Adresse zur Ausführung ansteht.

Es gibt einige Techniken, Unterprogramme ohne den exakten CALL-RET-Mechanismus zu benutzen. Bei geschachtelten Unterprogrammen könnte es vorkommen, daß der letzte Befehl des äußeren Unterprogramms ein CALL-Befehl ist; anschließend folgt dann der Rücksprung ins Hauptprogramm (oder in ein noch weiter außen liegendes Unterprogramm). Nehmen wir also an, daß wir zwei geschachtelte Unterprogramme UVW und XYZ haben, wobei UVW von einem Hauptprogramm, XYZ vom Unterprogramm UVW aufgerufen wird:

```
; Hauptprogramm

            :
            :
            :
            CALL        UVW         ; Unterprogramm UVW aufrufen
HPRA:                               ; Rueckkehradresse des
                                    ; Hauptprogramms
            :
            :
            :

; Unterprogramm UVW

UVW:
            :
            :
            :
            CALL        XYZ         ; Unterprogramm XYZ aufrufen
UPRA:       RET                     ; Rueckkehradresse des
                                    ; Unterprogramms UVW,
                                    ; Ruecksprung ins Hauptprogramm

; Unterprogramm XYZ

XYZ:
            :
            :
            :
            RET                     ; Ruecksprung
                                    ; ins Unterprogramm UVW
```

Nach dem Aufruf von XYZ sieht der Stapel folgendermaßen aus:

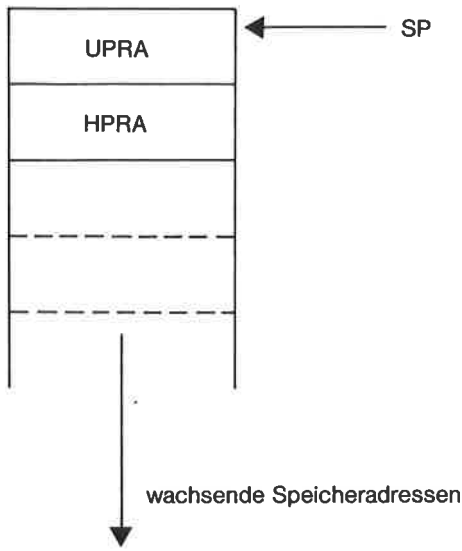


Bild 19.2. *Stapel bei geschachtelten Unterprogrammaufrufen*

Nach der Durchführung des RET-Befehls im Unterprogramm XYZ sieht der Stapel dann so aus:

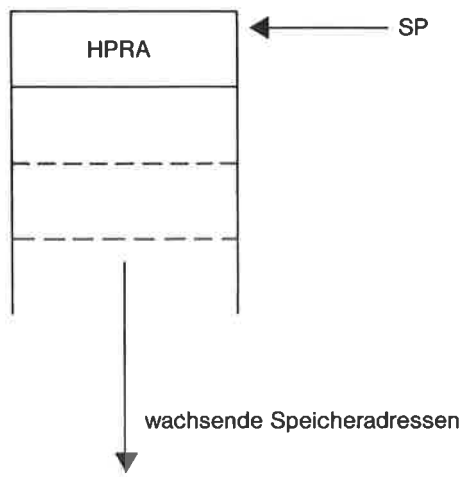


Bild 19.3. *Stapel nach Rückkehr aus dem inneren Unterprogramm*

Nun wird der RET-Befehl im Unterprogramm UVW ausgeführt, wodurch auch die Rückkehradresse des Hauptprogramms vom Stapel verschwindet und das Programm an der Adresse HPRA fortgesetzt wird. Da zwischen dem Unterprogramm-Aufruf CALL XYZ und dem RET-

Befehl im Unterprogramm UVW keine weiteren Befehle stehen, kann das Unterprogramm XYZ auch gleich ins Hauptprogramm zurückkehren. Wir ersetzen also die Sequenz

```
        CALL    XYZ
        RET
```

durch den Befehl

```
        JP      XYZ         ; Unterprogramm XYZ anspringen
```

Das Resultat ist ein kürzerer Objekt-Code und eine schwächere Belegung des Stapels.

Eine weitere Modifikation des Unterprogramm-Aufruf-Schemas besteht darin, die Rückkehradresse zu berechnen, sie auf den Stapel abzulegen und anschließend das Unterprogramm anzuspringen; dies ist sinnvoll, wenn nach der Ausführung des Unterprogramms nicht hinter dem CALL-Befehl fortgesetzt werden soll, sondern an einer anderen Stelle des Programms. Wir betrachten dazu folgendes Beispiel:

```
; Hauptprogramm

        :
        :
        :
HPRA:                           ; Rueckkehradresse
                                ; des Hauptprogramms
        :
        :
        :
        LD      HL,HPRA         ; Rueckkehradresse des
                                ; Hauptprogramms laden
        PUSH    HL              ; Rueckkehradresse auf den
                                ; Stapel bringen
        JP      UP              ; Unterprogramm UP anspringen
        :
        :
        :

; Unterprogramm UP

UP:
        :
        :
        :
        RET                     ; Ruecksprung ins Hauptprogramm
```

Tritt in einem Unterprogramm ein schwerwiegender Fehler auf, so ist es meist ratsam, das Unterprogramm abzubrechen und zu einer speziellen Fehlerbehandlungsroutine zu springen. Damit die (sinnlos gewordene) Rückkehradresse in diesem Fall nicht den Stapel verschmutzt, sollte sie entfernt werden, zum Beispiel durch Inkrementieren des Stapel-Zeigers:

```
; Hauptprogramm

                :
                :
                :
                CALL        UP          ; Unterprogramm UP aufrufen
                :
                :
                :

; Unterprogramm UP

UP:
                :
                :
                :
                JP          NZ,OK       ; weitermachen, falls kein
                                        ; Fehler aufgetreten
                INC         SP          ; Rueckkehradresse vom
                INC         SP          ; Stapel entfernen
                JP          FEHLER      ; Fehlerbehandlungsroutine
                                        ; anspringen
OK:
                :
                :
                :
                RET                     ; Rueckkehr ins Hauptprogramm
```

Das Unterprogramm kann die ihm mitgelieferte Rückkehradresse auch modifizieren, wenn es notwendig ist. Dazu wird die alte Rückkehradresse vom Stapel entfernt und dann die neue Rückkehradresse auf den Stapel gebracht; es bietet sich die Verwendung eines EX-Befehls an:

```
; Hauptprogramm

                :
                :
                :
HPRA:                                   ; Rueckkehradresse
                                        ; des Hauptprogramms
```

```
                :
                :
                :
                CALL        UP          ; Unterprogramm UP aufrufen
                :
                :
                :

; Unterprogramm UP

UP:
                :
                :
                :
                LD          HL,HPRA     ; Rueckkehradresse des
                                        ; Hauptprogramms laden
                EX          (SP),HL     ; alte Rueckkehradresse vom
                                        ; Stapel entfernen, neue
                                        ; Rueckkehradresse auf den
                                        ; Stapel legen
                :
                :
                :
                RET                     ; ins Hauptprogramm
                                        ; zurueckspringen
```

Ein häufig auftretender Fall, den wir im Unterkapitel »Stapel-Schnittstellen« intensiv studieren werden, liegt vor, wenn das Unterprogramm Parameter benötigt, die auf dem Stapel abgelegt wurden. Ein CALL-Befehl legt als oberstes Element des Stapels immer die Rückkehradresse ab, so daß das Unterprogramm zunächst nicht an die Parameter auf dem Stapel gelangen kann. Eine gängige Technik besteht darin, die Rückkehradresse in ein Register (vorwiegend ein Indexregister) zu holen und später indirekt anzuspringen; dies hat dieselbe Wirkung wie ein RET-Befehl, gibt aber zunächst einmal den Zugriff auf die Parameter frei. Beispiel:

```
; Hauptprogramm

                :
                :
                :
                PUSH        DE          ; Parameter auf den
                                        ; Stapel bringen
                CALL        UP          ; Unterprogramm UP aufrufen
                :
                :
                :
```

```
; Unterprogramm UP

UP:     POP     IX      ; Rueckkehradresse vom Stapel
                        ; nehmen und sichern
        POP     BC      ; Parameter vom Stapel holen
        :
        :
        :
        JP      (IX)    ; Ruecksprung ins Hauptprogramm
```

Übungen

1. Schreibe ein Unterprogramm für die Umwandlung einer ASCII-codierten Hex-Ziffer in ihre Binärcodierung. Setze damit ein Byte aus zwei ASCII-codierten Hex-Ziffern zusammen.

2. Ein Feld von Nibbles enthalte lauter binär-codierte Dezimalziffern. Bilde die Summe der Ziffern des Felds. Verwende dafür mehrere Unterprogramme.

3. Schreibe ein Programm, das feststellt, ob eine Zeichenkette eine ganze Dezimalzahl darstellt. Der Betrag einer ganzen Dezimalzahl ist eine lückenlose Folge von Dezimalziffern. Vor dem Betrag kann ein Vorzeichen stehen (+ oder −); dabei dürfen zwischen Vorzeichen und Betrag auch Leerzeichen stehen. Vor und hinter der Zahl dürfen ebenfalls beliebig viele Leerzeichen stehen.

19.2 Seiteneffekte

Kommen wir noch einmal auf das Unterprogramm HEXASC zurück. In diesem Unterprogramm wurde der Inhalt des A-Registers verändert; diese Veränderung war beabsichtigt und notwendig, da wir das ASCII-Zeichen ja im A-Register zurückliefern wollen. Allerdings tritt durch das Unterprogramm eine weitere Veränderung eines Registers ein, die weder beabsichtigt noch notwendig ist, die sogar unter Umständen störend wirken kann: die Zerstörung der Flags (F-Register). Eine solche nicht beabsichtigte Veränderung eines Registers oder einer Speicherzelle nennt man einen *Seiteneffekt.*

Eine Veränderung der Flags durch einen Seiteneffekt wird meist in Kauf genommen. Im Falle des Unterprogramms HEXASC läßt sich anhand der Flags sogar feststellen, ob das berechnete Zeichen eine Dezimalziffer oder ein Hex-Buchstabe ist; man wird deshalb unter Umständen sogar bewußt von diesem Seiteneffekt Gebrauch machen. Ein weiteres Beispiel für einen erwünschten Seiteneffekt machen wir an folgendem Unterprogramm für das Kopieren eines Bytes deutlich:

```
KOPIE:  LD      A,(HL)      ; Byte holen
        LD      (DE),A      ; Byte kopieren
        INC     HL          ; Zeiger
        INC     DE          ; fortschalten
        RET                 ; Ruecksprung
```

Der erwünschte Seiteneffekt besteht in der Fortschaltung der beiden Zeiger; das Unterprogramm kann nun durch eine Schleife auf eine Folge von Bytes angewandt werden. Im allgemeinen wird man jedoch versuchen, möglichst ohne Seiteneffekte zu programmieren; denn Seiteneffekte erhöhen die Fehleranfälligkeit von Programmen.

Für den Umgang mit Seiteneffekten gibt es zwei Prinzipien. Die erste Methode besteht darin, die Seiteneffekte eines Unterprogramms zu dokumentieren und dem aufrufenden Programm alle Vorsorgemaßnahmen zur Sicherung wichtiger Daten zu überlassen. Die zweite Möglichkeit überträgt die volle Verantwortung auf das Unterprogramm; dieses muß dann dafür sorgen, daß von Zerstörung bedrohte Daten zunächst gesichert und später restauriert werden, vorzugsweise mit Hilfe des Stapels. (Eine dritte Möglichkeit wird anscheinend von den Entwicklern von Betriebssystemen gern verwendet, nämlich alle Register nach besten Kräften zu zerstören, dem Anwender dies aber nicht mitzuteilen!)

Beide Methoden haben Vor- und Nachteile. Für die erste Methode spricht, daß das aufrufende Programm ja am besten weiß, welche Daten es später noch benötigt, und daß somit nicht unnötig viele Daten gesichert werden. Die zweite Methode hat den Vorteil, daß die Sicherungsmaßnahmen an einer zentralen Stelle – im Unterprogramm – stehen; dies reduziert die Länge des für Sicherungen aufgewandten Objekt-Codes und ist weniger fehleranfällig. Außerdem wird durch die zweite Methode das Innenleben des Unterprogramms vor der Außenwelt abgeschirmt; dies erleichtert die modulare Programmierung. Ich bevorzuge deshalb die zweite Möglichkeit. Ein Beispiel hierzu: Vertauschung der Inhalte des B-Registers und des C-Registers.

```
TAUSCH: LD      A,B         ; Inhalt von B hilfsweise sichern
        LD      B,C         ; Inhalt von C nach B kopieren
        LD      C,A         ; alten Inhalt von B
                            ; nach C kopieren
        RET                 ; Ruecksprung
```

Als Seiteneffekt wird hierbei das A-Register zerstört. Durch Sichern des AF-Registers auf dem Stapel wird unser Unterprogramm seiteneffektfrei:

```
TAUSCH: PUSH    AF          ; A-Register sichern
        LD      A,B         ; Inhalt von B hilfsweise sichern
        LD      B,C         ; Inhalt von C nach B kopieren
        LD      C,A         ; alten Inhalt von B
                            ; nach C kopieren
        POP     AF          ; A-Register restaurieren
        RET                 ; Ruecksprung
```

Zu den möglichen Seiteneffekten gehört auch die Verschmutzung des Stapels beziehungsweise ein falsch gesetzter Stapel-Zeiger nach Rückkehr aus einem Unterprogramm durch einen Sprung. Auch eine Veränderung global genutzter Speicherstellen oder Ausgaben auf dem Bildschirm werden im weiteren Sinne als Seiteneffekte bezeichnet.

Übungen

1. Schreibe ein Unterprogramm ohne Seiteneffekte, das den Inhalt des A-Registers - interpretiert als vorzeichenlose ganze Zahl - halbiert, wobei der Rest der Division irrelevant ist.

2. Worin bestehen die Seiteneffekte bei folgendem Unterprogramm zum Kopieren eines Felds?

```
KOPIE:  LD      A,B     ; Laenge auf Null
        OR      C       ; pruefen
        RET     Z       ; Laenge Null, nichts zu kopieren
        LDIR            ; Feld kopieren
        RET             ; Ruecksprung
```

19.3 Register-Schnittstellen

Ein Programm erhält seine Eingabedaten vom Benutzer und gibt diesem die Resultate zurück; in ähnlicher Form muß auch ein Unterprogramm vom aufrufenden Programm Eingabedaten - sogenannte *Parameter* - erhalten und ihm Resultate zurückliefern. Die Art und Weise, in der die Parameter übergeben und die Resultate zurückgegeben werden, nennt man die *Schnittstelle* (engl. interface) des Unterprogramms. Bei modularer Programmierung ist die Beschreibung der Schnittstelle eines Unterprogramms die einzige Information, die dem Benutzer des Unterprogramms zugänglich ist. Natürlich gehört zu einer Schnittstellenbeschreibung auch eine Darstellung des funktionellen Zusammenhangs zwischen Parametern und Ergebnissen. Wie eine sinnvolle Schnittstellen-Beschreibung für Unterprogramme des Z80 auszusehen hat, werden wir im Unterkapitel »Dokumentation von Schnittstellen« erläutern.

Es gibt verschiedene Arten von Schnittstellen. Eine davon haben wir im Unterprogramm HEXASC bereits kennengelernt: eine *Register-Schnittstelle*. Ein Unterprogramm mit (reiner) Register-Schnittstelle erhält seine Parameter als Register-Inhalte und gibt seine Ergebnisse ebenfalls als Register-Inhalte an das aufrufende Programm zurück. Die Flags gehören selbstverständlich auch zu den Registern. Auch das Unterprogramm TAUSCH verfügt über eine Register-Schnittstelle.

Wenn die Parameter komplizierte Strukturen sind (Verbunde, Zeichenketten, Gleitpunkt-Zahlen), passen sie natürlich nicht in ein Register. Man kann dann eine solche Struktur auf mehrere Register verteilen; Gleitpunkt-Zahlen einfacher Genauigkeit (siehe das Kapitel

»Gleitpunkt-Zahlen«) benötigen vier Bytes und lassen sich zum Beispiel im Superregister HL & DE unterbringen.

Die Verwendung von Register-Schnittstellen ist durch die Zahl und den Aufbau der vorhandenen Register prinzipiell limitiert, da dabei alle Parameter beziehungsweise alle Ergebnisse gleichzeitig in Registern stehen müssen.

Der Zugriff auf die Parameter und das Rückgeben der Ergebnisse erfolgt bei Register-Schnittstellen zeit- und speicherökonomisch. Diese Sichtweise bezieht sich jedoch nur auf die Verhältnisse innerhalb des Unterprogramms; möglicherweise stehen im aufrufenden Programm die Parameter bereits im Speicher und müssen nun zuerst in Register übertragen werden.

Ein prinzipieller Vorteil von Register-Schnittstellen ist ihre Unabhängigkeit vom Datenspeicher, insbesondere vom Stapel. Die Bearbeitung der Parameter kann es allerdings notwendig machen, einen Teil der Parameter innerhalb des Unterprogramms auf dem Stapel zu sichern.

Übungen

1. Schreibe ein Unterprogramm, das abhängig von einer Funktionsnummer im A-Register folgendes durchführt:
   - Addieren zweier Zahlen
   - Subtrahieren zweier Zahlen
   - Absoluter Betrag einer Zahl
   - Negieren einer Zahl

   Überlegen Sie sich hierzu selbst eine geeignete Schnittstelle.

2. Überlegen Sie sich für folgende Aufgabe eine passende Schnittstelle und ein Unterprogramm: Vergleiche zwei vorzeichenlose ganze 32-Bit-Zahlen und liefere das Ergebnis (<, =, >) zurück.

19.4 Speicher-Schnittstellen

Eine weitere Form von Schnittstelle stellen die Speicher-Schnittstellen dar. Die Parameter und Ergebnisse stehen dabei in bestimmten Speicherzellen, die sowohl dem aufrufenden Programm wie auch dem Unterprogramm bekannt sein müssen. Die Speicherzellen können dem Unterprogramm fest zugeordnet sein, was bedeutet, daß sie ausschließlich dem Unterprogramm zur Bearbeitung zur Verfügung stehen (mit Ausnahme des Hineingebens der Parameter und des Abholens der Ergebnisse). Zum Beispiel bildet folgendes Unterprogramm die Summe von zwei 32-Bit-Zahlen in 2-Komplement-Darstellung, die es in seinen lokalen Variablen LIOP und REOP vorzufinden erwartet, und legt die Summe – wieder als 32-Bit-Größe – in seiner lokalen Variablen ERGEBN ab (um Seiteneffekte kümmern wir uns momentan nicht):

```
; Datenbereich

LIOP:     DEFS     4          ; lokale Variable
                              ; des Unterprogramms ADD32:
                              ; linker Operand der Addition
REOP:     DEFS     4          ; lokale Variable
                              ; des Unterprogramms ADD32:
                              ; rechter Operand der Addition
ERGEBN:   DEFS     4          ; lokale Variable
                              ; des Unterprogramms ADD32:
                              ; Ergebnis der Addition

; Programmbereich

ADD32:    LD       HL,(LIOP)      ; niederwertigen Anteil der
          LD       DE,(REOP)      ; beiden Operanden addieren
          ADD      HL,DE          ; und als niederwertigen Anteil
          LD       (ERGEBN),HL    ; des Ergebnisses abspeichern
          LD       HL,(LIOP+2)    ; hoeherwertigen Anteil der
          LD       DE,(REOP+2)    ; beiden Operanden addieren,
          ADC      HL,DE          ; Uebertrag beruecksichtigen und
          LD       (ERGEBN+2),HL  ; hoeherwertigen Anteil des
                                  ; Ergebnisses abspeichern
          RET                     ; Ruecksprung
```

Die Speicherzellen einer Speicher-Schnittstelle können aber auch vielen Programmen gemeinsam gehören; alle diese Programme schreiben dann ihre Parameter in dieselben Speicherzellen und holen ihre Ergebnisse aus denselben Speicherzellen ab. Sinnvoll wird dieses Verfahren, wenn wir es mit einer Gruppe zusammengehöriger Unterprogramme zu tun haben, die ähnliche Funktionen auf Parametern von gleicher Struktur ausführen, beispielsweise die arithmetischen Operationen +, −, *, / auf Gleitpunkt-Zahlen.

Häufig tritt eine Kombination von Register-Schnittstelle und Speicher-Schnittstelle auf. Die eigentlichen Parameter stehen dabei in einem oder mehreren Parameterblöcken (zusammenhängenden Speicherbereichen). Dem Unterprogramm sind die Adressen dieser Speicherbereiche nicht a priori bekannt; die Zeiger werden in Registern als Hilfsparameter an das Unterprogramm übergeben, das mit Hilfe der Registerinhalte nun auf die Parameterblöcke zugreifen kann. Ein Beispiel für diese Art von Schnittstelle finden wir im Unterprogramm KOPIE aus Übung 2 des Unterkapitels »Seiteneffekte«; das HL-Register zeigt dabei auf einen der eigentlichen Parameter, auf ein Feld von Bytes (ein weiterer eigentlicher Parameter, die Länge des Felds, wird in einem Register übergeben; die Ablageadresse für das Ergebnis wird als Zeiger ebenfalls in einem Register übergeben).

Das Übergeben eines Zeigers auf einen Parameter, Parameterblock oder den Speicherbereich eines Ergebnisses hat den (besonders bei großen Datenstrukturen hoch einzuschätzen-

den) Vorteil, daß der Zeiger im Gegensatz zur Datenstruktur selbst eine feste und kurze Länge hat, und daß ein Kopieren der gesamten Datenstruktur sich erübrigt.

Mittels einer Speicher-Schnittstelle oder kombinierten Register-/Speicher-Schnittstelle können beliebig viele und beliebig große Datenstrukturen zwischen Unterprogramm und aufrufendem Programm ausgetauscht werden. Während bei der Übergabe von Zeigern diese meist zur indirekten Adressierung der Parameter benutzt werden können und damit Zugriffszeiten wie eine Register-Schnittstelle ermöglichen, muß bei einer Speicher-Schnittstelle mit längeren Zugriffszeiten gerechnet werden. Da reine Speicher-Schnittstellen zudem recht unflexibel sind (zum Beispiel führen sie zu Problemen beim rekursiven Aufruf von Unterprogrammen, siehe die Unterkapitel »Eintritts-invariante Unterprogramme« und »Rekursive Unterprogramme«), werden sie selten verwendet.

Übungen

1. Schreibe ein Unterprogramm, das die Länge einer Zeichenkette mit Endemarkierung berechnet; vermeide dabei Seiteneffekte.

2. Löse die Aufgabe 2 aus dem Unterkapitel 19.3 für zwei 64-Bit-Zahlen. Vergleiche die beiden Programme!

19.5 Stapel-Schnittstellen

Die kompliziertesten, dafür aber auch flexibelsten Schnittstellen sind die *Stapel-Schnittstellen*. Bei einer (reinen) Stapel-Schnittstelle werden die Parameter vor dem Aufruf des Unterprogramms auf den Stapel gebracht, wo sie vom Unterprogramm mit den uns bekannten Methoden der Stapel-Bearbeitung abgeholt werden können; umgekehrt erwartet das aufrufende Programm nach Rückkehr aus dem Unterprogramm seine Ergebnisse auf dem Stapel. In beiden Fällen wird die Bearbeitung erschwert durch die Tatsache, daß direkt nach einem CALL-Befehl beziehungsweise direkt vor einem RET-Befehl die Rückkehradresse das oberste Stapel-Element sein muß.

Wenn das Unterprogramm die Parameter vom Stapel entfernen will, so muß zuerst die Rückkehradresse gerettet werden. Am besten gefällt mir die Technik, die Rückkehradresse in ein Indexregister zu holen und am Ende des Unterprogramms statt eines RET-Befehls einen indirekten Sprung auszuführen (siehe Kapitel »Der Stapel«). Die zweite gängige Methode besteht darin, die Rückkehradresse in ein beliebiges Register zu holen, alle Parameter vom Stapel ins Register zu bringen und die Rückkehradresse dann wieder auf den Stapel zu legen. Dazu ein Beispiel mit vier Parametern vom Typ Wort:

```
UP:     POP     HL      ; Rueckkehradresse sichern
        POP     BC      ; 1. Parameter holen
        POP     DE      ; 2. Parameter holen
```

```
        POP     IX      ; 3. Parameter holen
        POP     IY      ; 4. Parameter holen
        PUSH    HL      ; Rueckkehradresse wieder
                        ; korrekt auf dem Stapel ablegen
        :
        :
        :
```

Bei beiden Methoden sind nach der Rückkehr ins aufrufende Programm die Parameter vom Stapel verschwunden. Sollen Ergebnisse auf dem Stapel abgelegt werden, so müssen bei Methode 1 zunächst alle Parameter abgebaut werden; bei Methode 2 wenden wir wieder dasselbe Prinzip wie oben an, nämlich: Rückkehradresse sichern, Ergebnisse auf den Stapel bringen, Rückkehradresse wieder auf dem Stapel ablegen.

Manchmal möchte man Unterprogramme schreiben, die eine beliebige Anzahl von Parametern gleichen Typs bearbeiten können, zum Beispiel für das Konkatenieren einer beliebigen Anzahl von Zeichenketten, für das Summieren einer Folge von Zahlen oder für das Verknüpfen einer Menge von Inzidenzvektoren (es gibt noch viele weitere Beispiele). Dann ist es notwendig, dem Unterprogramm die Anzahl der Parameter mitzuteilen. Die Anzahl kann in einem Register stehen. Sie kann aber auch als oberster Parameter auf dem Stapel liegen; wir betrachten dazu ein Unterprogramm, das eine beliebige Anzahl von Worten addiert:

```
ADD:    POP     IX      ; Rueckkehradresse sichern
        POP     BC      ; Anzahl der Parameter holen
        LD      HL,0    ; Akkumulator loeschen
TEST:   LD      A,B     ; sind alle Parameter
        OR      C       ; abgearbeitet?
        JP      Z,FERTIG ; alle Parameter abgearbeitet
        POP     DE      ; naechsten Parameter holen
        ADD     HL,DE   ; und verwenden
        DEC     BC      ; restliche Anzahl
                        ; von Parametern berechnen
        JP      TEST    ; pruefen, ob weitere
                        ; Parameter folgen
FERTIG: PUSH    HL      ; Ergebnis ablegen
        JP      (IX)    ; Ruecksprung
```

Genauso kann ein Unterprogramm, das eine beliebige Anzahl von Ergebnissen auf dem Stapel ablegt, durch das oberste Stapel-Element dem aufrufenden Programm die Anzahl der Ergebnisse bekanntgeben.

Bei der Reihenfolge der Parameter ist eines zu beachten: Bringt das aufrufende Programm mittels PUSH-Befehlen der Reihe nach den ersten, zweiten, ..., vorletzten, letzten Parameter auf den Stapel, so erhält das Unterprogramm durch sukzessive POP-Befehle den letzten, vorletzten, ..., zweiten, ersten Parameter geliefert – und umgekehrt!

Ein ganz anderer Zugang besteht darin, den Stapel-Zeiger dort zu lassen, wo er beim Eintritt in das Unterprogramm stand, und die Parameter indirekt als Verbund oder als Feld zu adressieren (siehe Kapitel »Der Stapel«). Nach Rückkehr ins aufrufende Programm ist es dessen Aufgabe, den Stapel von den nutzlos gewordenen Parametern zu säubern, zum Beispiel folgendermaßen:

```
LD      IY,0        ; Stapel-Zeiger in IY
ADD     IY,SP       ; sichern
PUSH    BC          ; Parameter auf den Stapel werfen
PUSH    DE          ; Parameter auf den Stapel werfen
PUSH    HL          ; Parameter auf den Stapel werfen
CALL    UP          ; Unterprogramm UP aufrufen
LD      SP,IY       ; Stapel-Zeiger restaurieren
```

Etwas problematisch ist dabei, daß Ergebnisse nur dann auf dem Stapel zurückgeliefert werden können, wenn durch sie auf dem Stapel befindliche Parameter überschrieben werden. Das aufrufende Programm kann aber den Parameter-Block um Speicherplätze für die Ergebnisse erweitern, indem es den Stapel-Zeiger vor dem Unterprogramm-Aufruf entsprechend dekrementiert:

```
LD      IY,0        ; Stapel-Zeiger in IY
ADD     IY,SP       ; sichern
PUSH    BC          ; Parameter auf den Stapel werfen
PUSH    DE          ; Parameter auf den Stapel werfen
PUSH    HL          ; Parameter auf den Stapel werfen
DEC     SP          ; Platz
DEC     SP          ; fuer
DEC     SP          ; Ergebnisse
DEC     SP          ; reservieren
CALL    UP          ; Unterprogramm UP aufrufen
POP     DE          ; Ergebnis vom Stapel holen
POP     IX          ; Ergebnis vom Stapel holen
LD      SP,IY       ; Stapel-Zeiger restaurieren
```

Stapel-Schnittstellen werden intensiv von den Compilern höherer Programmiersprachen (zum Beispiel PASCAL) benutzt; auch rekursive Programme besitzen fast ausschließlich Stapel-Schnittstellen.

Übungen

1. Schreibe ein Unterprogramm, das eine beliebige Folge von Teilmengen einer 8elementigen Menge zu einer Gesamtmenge vereinigt. Die Anzahl der Teilmengen und die Inzidenzvektoren der Teilmengen sollen auf dem Stapel übergeben werden; die Vereinigungsmenge soll auf dem Stapel zurückgeliefert werden.

19.6 Dokumentation von Schnittstellen

Schreiben wir ein Unterprogramm nicht nur für den einmaligen Gebrauch, sondern zur dauernden Verwendung in einer Programm-Bibliothek, so muß eine knappe, aber vollständige Dokumentation darüber erstellt werden. Diese besteht aus folgenden Angaben:

- Name des Unterprogramms
- Funktion des Unterprogramms
- Parameter des Unterprogramms
- Ergebnisse des Unterprogramms
- Seiteneffekte

Wir demonstrieren dies an einigen Beispielen aus den vorangegangenen Unterkapiteln:

```
; Name:           HEXASC
; Funktion:       Umwandlung einer Hex-Ziffer, die binaer codiert
;                 ist, in die entsprechende ASCII-Codierung
; Parameter:      A-Register:   niederwertiger Nibble enthaelt
;                               binaer-codierte Hex-Ziffer,
;                               hoeherwertiger Nibble ist Null
; Ergebnis:       A-Register:   ASCII-Codierung der Hex-Ziffer
; Seiteneffekt:   F-Register zerstoert,
;                 Uebertrag-Flag gesetzt, falls Hex-Ziffer eine
;                 Dezimalziffer ist

HEXASC:   ADD     '0'          ; Dezimalziffer codieren
          CP      '0'+10       ; testen, ob es eine
                               ; Dezimalziffer war
          RET     C            ; es war eine Dezimalziffer,
                               ; Codierung durchgefuehrt,
                               ; Ruecksprung
          ADD     'A'-0AH-'0'  ; Korrektur fuer Buchstaben
          RET                  ; Ruecksprung
```

```
; Name:           KOPIE
; Funktion:       kopiert ein Byte eines Felds in ein anderes Feld
; Parameter:      HL-Register:  Zeiger auf zu kopierendes Byte
;                 DE-Register:  Zeiger auf Speicherplatz, in den
;                               das Byte kopiert werden soll
; Ergebnis:       implizit
; Seiteneffekt:   HL-Register wird um 1 erhoeht
;                 DE-Register wird um 1 erhoeht
;                 A-Register enthaelt kopiertes Zeichen
```

```
KOPIE:  LD      A,(HL)      ; Byte holen
        LD      (DE),A      ; Byte kopieren
        INC     HL          ; Zeiger
        INC     DE          ; fortschalten
        RET                 ; Ruecksprung
```

```
; Name:          TAUSCH
; Funktion:      vertauschen der Inhalte des B-Registers und
;                C-Registers
; Parameter:     B-Register
;                C-Register
; Ergebnis:      implizit
; Seiteneffekt:
```

```
TAUSCH: PUSH    AF          ; A-Register sichern
        LD      A,B         ; Inhalt von B hilfsweise sichern
        LD      B,C         ; Inhalt von C nach B kopieren
        LD      C,A         ; alten Inhalt von B
                            ; nach C kopieren
        POP     AF          ; A-Register restaurieren
        RET                 ; Ruecksprung
```

```
; Name:          ADD32
; Funktion:      Addition zweier 32-Bit-Groessen in 2-Komplement-
;                Darstellung (ohne Beruecksichtigung eines
;                Ueberlaufs)
; Parameter:     Speicherzellen mit den Adressen LIOP bis LIOP+3:
;                linker Operand in Low/High-Darstellung
;                Speicherzellen mit den Adressen REOP bis REOP+3:
;                rechter Operand in Low/High-Darstellung
; Ergebnis:      Speicherzellen mit den Adressen ERGEBN bis
;                ERGEBN+3: Summe in Low/High-Darstellung
; Seiteneffekt:  F-Register zerstoert,
;                Ueberlauf-Flag zeigt Ueberlauf an

; Datenbereich

LIOP:   DEFS    4           ; lokale Variable
                            ; des Unterprogramms ADD32:
                            ; linker Operand der Addition
```

```
REOP:     DEFS      4          ; lokale Variable
                               ; des Unterprogramms ADD32:
                               ; rechter Operand der Addition
ERGEBN:   DEFS      4          ; lokale Variable
                               ; des Unterprogramms ADD32:
                               ; Ergebnis der Addition

; Programmbereich

ADD32:    LD        HL,(LIOP)  ; niederwertigen Anteil der
          LD        DE,(REOP)  ; beiden Operanden addieren
          ADD       HL,DE      ; und als niederwertigen Anteil
          LD        (ERGEBN),HL ; des Ergebnisses abspeichern
          LD        HL,(LIOP+2) ; hoeherwertigen Anteil der
          LD        DE,(REOP+2) ; beiden Operanden addieren,
          ADC       HL,DE      ; Uebertrag beruecksichtigen und
          LD        (ERGEBN+2),HL ; hoeherwertigen Anteil des
                               ; Ergebnisses abspeichern
          RET                  ; Ruecksprung
```

```
; Name:          ADD
; Funktion:      Addition einer beliebigen Anzahl von Worten
;                ohne Beruecksichtigung eines Uebertrags
; Parameter:     Stapel:  Anzahl n der Worte
;                         Wort 1
;                         :
;                         Wort n
; Ergebnis:      Stapel:  Summe
; Seiteneffekt:  AF-Register zerstoert
;                BC-Register zerstoert
;                DE-Register zerstoert
;                HL-Register zerstoert
;                IX-Register zerstoert
;                Parameter vom Stapel entfernt,
;                Ergebnis auf den Stapel gebracht

ADD:      POP       IX         ; Rueckkehradresse sichern
          POP       BC         ; Anzahl der Parameter holen
          LD        HL,0       ; Akkumulator loeschen
TEST:     LD        A,B        ; sind alle Parameter
          OR        C          ; abgearbeitet?
          JP        Z,FERTIG   ; alle Parameter abgearbeitet
```

```
        POP     DE      ; naechsten Parameter holen
        ADD     HL,DE   ; und verwenden
        DEC     BC      ; restliche Anzahl
                        ; von Parametern berechnen
        JP      TEST    ; pruefen, ob weitere
                        ; Parameter folgen
FERTIG: PUSH    HL      ; Ergebnis ablegen
        JP      (IX)    ; Ruecksprung
```

Wofür der Anwender eines so dokumentierten Unterprogramms selbst zu sorgen hat, ist natürlich das Übergeben korrekter Parameter; die Eigenschaften des Unterprogramms, die von der Schnittstellen-Beschreibung garantiert werden, beziehen sich nur auf Daten, welche die unter dem Punkt »Parameter« beschriebene Form haben.

Übungen

1. Dokumentiere die Unterprogramme, die in den Übungen dieses Kapitels vorkamen.

19.7 Betriebssystem-Schnittstellen

Betriebssysteme haben unter anderem die Aufgabe, dem Benutzer die Steuerung der Hardware zu erleichtern, indem sie eine Basis-Schicht von Unterprogrammen bereitstellen, die wichtige Ein-/Ausgabe-Funktionen durchführen. Die dem Benutzer vom Entwickler des Betriebssystems bekanntgemachten Unterprogramme bilden in ihrer Gesamtheit die Betriebssystem-Schnittstelle. Wir wollen uns im folgenden mit der des Betriebssystems CP/M beschäftigen.

Das Betriebssystem CP/M war lange Zeit das wohl am weitesten verbreitete Betriebssystem für Mikrocomputer mit Z80- oder 8080-Prozessor. Neuere Entwicklungen haben zu Betriebssystemen geführt, die einen Teil der Funktionen von CP/M enthalten oder nachahmen (Genaueres können Sie den Betriebssystem-Handbüchern Ihres Rechners entnehmen).

Wir werden einen Teil der unter CP/M zur Verfügung stehenden Unterprogramme hier dokumentieren; es geht dabei nicht um eine vollständige Darstellung (die Floppy-Disk-Operationen lassen wir ganz weg, sie sind in diesem Zusammenhang zu kompliziert), sondern eher darum, sich unter einer Betriebssystem-Schnittstelle überhaupt etwas vorstellen zu können.

Charakteristisch für die System-Schnittstelle von CP/M ist, daß über die eigentlichen Unterprogramme ein Verteilerprogramm gestülpt wurde, so daß der Benutzer die Unterprogramme nur über dieses Verteilerprogramm erreichen kann. Die Anfangsadresse des Verteilerprogramms ist üblicherweise 0005H; sie wird meist mit BDOS bezeichnet:

```
BDOS:   EQU     0005H   ; Adresse des
                        ; CP/M Verteilerprogramms
```

Die auszuführende Funktion wird über eine Funktionsnummer ausgewählt, die beim Aufruf im C-Register stehen muß. Weitere Parameter von BDOS werden im E-Register oder DE-Register übergeben. Ergebnisse werden im A-Register oder HL-Register zurückgeliefert.

Weil es manchmal nicht im Handbuch steht, obwohl es bitter notwendig wäre, schreibe ich hier eine Warnung: Die meisten Funktionen von CP/M zerstören als Seiteneffekt irgendwelche Register!

Nun zu den Funktionen selbst:

Das Unterprogramm mit der Funktionsnummer 00H führt einen Warmstart des Computers durch; es benötigt keine Parameter und hat kein Ergebnis.

```
        LD      C,00H       ; Funktionsnummer laden
        CALL    BDOS        ; Warmstart durchfuehren
```

Das Unterprogramm mit der Funktionsnummer 01H liest ein ASCII-Zeichen von der Tastatur ein; das Zeichen wird im A-Register zurückgeliefert. Das Unterprogramm benötigt keine Parameter. Es wird so lange gewartet, bis auf der Tastatur ein Zeichen eingegeben wird.

```
        LD      C,01H       ; Funktionsnummer laden
        CALL    BDOS        ; Zeichen von der Tastatur
                            ; ins A-Register holen
```

Das Unterprogramm mit der Funktionsnummer 02H gibt ein ASCII-Zeichen auf den Bildschirm aus; das Zeichen wird dabei im E-Register übergeben. Das Unterprogramm liefert kein Ergebnis.

```
        LD      E,'*'       ; Zeichen laden
        LD      C,02H       ; Funktionsnummer laden
        CALL    BDOS        ; Zeichen auf den Bildschirm
                            ; ausgeben
```

Mit Hilfe der beiden letzten Funktionen können wir uns eine Echo-Funktion schreiben, die von der Tastatur Zeichen einliest und auf dem Bildschirm wiedergibt:

```
FNEIN:  EQU     01H         ; Funktionsnummer fuer Eingabe
                            ; von der Tastatur
FNAUS:  EQU     02H         ; Funktionsnummer fuer Ausgabe
                            ; auf den Bildschirm
ECHO:   LD      C,FNEIN     ; Funktionsnummer fuer Eingabe
                            ; von der Tastatur laden
        CALL    BDOS        ; Zeichen von der Tastatur
                            ; ins A-Register holen
        LD      E,A         ; Zeichen in Parameter-Register
                            ; bringen
```

```
        LD      C,FNAUS     ; Funktionsnummer fuer Ausgabe
                            ; auf den Bildschirm laden
        CALL    BDOS        ; Zeichen auf den Bildschirm
                            ; ausgeben
        JP      ECHO        ; endlose Schleife bauen
```

Das Unterprogramm mit der Funktionsnummer 05H gibt ein ASCII-Zeichen auf den Drucker aus; das Zeichen wird dabei im E-Register übergeben. Das Unterprogramm liefert kein Ergebnis.

```
        LD      E,'*'       ; Zeichen laden
        LD      C,05H       ; Funktionsnummer laden
        CALL    BDOS        ; Zeichen auf den Drucker
                            ; ausgeben
```

Nun können wir unsere Echo-Funktion dahingehend modifizieren, daß jedes eingelesene Zeichen sowohl auf den Bildschirm als auch auf den Drucker ausgegeben wird:

```
FNEIN:  EQU     01H         ; Funktionsnummer fuer Eingabe
                            ; von der Tastatur
FNAUS:  EQU     02H         ; Funktionsnummer fuer Ausgabe
                            ; auf den Bildschirm
FNDRU:  EQU     05H         ; Funktionsnummer fuer Ausgabe
                            ; auf den Drucker
ECHO:   LD      C,FNEIN     ; Funktionsnummer fuer Eingabe
                            ; von der Tastatur laden
        CALL    BDOS        ; Zeichen von der Tastatur
                            ; ins A-Register holen
        LD      E,A         ; Zeichen in Parameter-Register
                            ; bringen
        LD      C,FNAUS     ; Funktionsnummer fuer Ausgabe
                            ; auf den Bildschirm laden
        PUSH    DE          ; Zeichen sichern!
        CALL    BDOS        ; Zeichen auf den Bildschirm
                            ; ausgeben
        POP     DE          ; Zeichen restaurieren
        LD      C,FNDRU     ; Funktionsnummer fuer Ausgabe
                            ; auf den Drucker laden
        CALL    BDOS        ; Zeichen auf den Drucker
                            ; ausgeben
        JP      ECHO        ; endlose Schleife bauen
```

Mit dem Unterprogramm der Funktionsnummer 0BH können wir den Tastatur-Status abfra-

gen. Das Unterprogramm benötigt keine Parameter; als Ergebnis enthält das A-Register den Wert FFH, falls ein Zeichen an der Tastatur zur Abholung ansteht, sonst den Wert 00H.

```
        LD      C,0BH       ; Funktionsnummer laden
        CALL    BDOS        ; Tastatur-Status ins
                            ; A-Register holen
```

Wir bauen mit Hilfe dieser Funktion ein Unterprogramm, das so lange Fragezeichen auf den Bildschirm ausgibt, bis an der Tastatur eine Dezimalziffer eingegeben wird:

```
FNEIN:  EQU     01H         ; Funktionsnummer fuer Eingabe
                            ; von der Tastatur
FNAUS:  EQU     02H         ; Funktionsnummer fuer Ausgabe
                            ; auf den Bildschirm
FNSTAT: EQU     0BH         ; Funktionsnummer fuer Abfragen
                            ; des Tastatur-Status
ZIFFER: LD      E,'?'       ; auszugebendes Zeichen laden
        LD      C,FNAUS     ; Funktionsnummer fuer Ausgabe
                            ; auf den Bildschirm laden
        CALL    BDOS        ; Fragezeichen ausgeben
        LD      C,FNSTAT    ; Funktionsnummer fuer Abfragen
                            ; des Tastatur-Status laden
        CALL    BDOS        ; Tastatur-Status ins A-Register
        OR      A           ; holen und auf Null pruefen
        JP      Z,ZIFFER    ; noch kein Zeichen eingegeben
        LD      C,FNEIN     ; Funktionsnummer fuer Eingabe
                            ; von der Tastatur laden
        CALL    BDOS        ; Zeichen von der Tastatur
                            ; ins A-Register holen
        CP      '0'         ; pruefen, ob es
        JP      C,ZIFFER    ; eine
        CP      '9'+1       ; Dezimalziffer
        JP      NC,ZIFFER   ; ist
        RET                 ; Dezimalziffer im A-Register
```

Das Unterprogramm mit der Funktionsnummer 09H dient der Ausgabe einer Zeichenkette auf den Bildschirm; die Zeichenkette muß durch ein Dollarzeichen »$« abgeschlossen sein. Im DE-Register wird ein Zeiger auf die Zeichenkette übergeben. Das Unterprogramm hat kein Ergebnis.

```
; Datenbereich

KETTE:  'Was ist los?$'     ; Zeichenkette
                            ; mit Endemarkierung $
```

```
; Programmbereich

        LD      DE,KETTE    ; Zeiger auf Zeichenkette laden
        LD      C,09H       ; Funktionsnummer fuer Ausgabe
                            ; einer Zeichenkette auf
                            ; Bildschirm laden
        CALL    BDOS        ; Zeichenkette auf Bildschirm
                            ; ausgeben
```

Übungen

1. Schreibe ein Programm, das eine Zeichenkette mit Längenangabe auf den Bildschirm bringt.

2. Schreibe ein Programm, das eine Zeichenkette von der Tastatur einliest.

3. Schreibe ein Unterprogramm, das ein ASCII-Zeichen von der Tastatur einliest, seinen ASCII-Code in zwei Hex-Ziffern konvertiert und diese auf den Bildschirm ausgibt.

4. Schreibe ein Programm, welches permanent das jeweils letzte auf der Tastatur gedrückte Zeichen auf den Bildschirm ausgibt.

19.8 Rekursive Unterprogramme

Eine spezielle Klasse von Unterprogrammen sind die *rekursiven Unterprogramme.* Ein rekursives Unterprogramm ist dadurch charakterisiert, daß es sich selbst aufruft; dies ergibt natürlich nur dann einen Sinn, wenn dies mit geänderten Parameterwerten geschieht. Rekursive Unterprogramme braucht man zum Beispiel bei der Analyse mathematischer Ausdrücke, bei der Bearbeitung von Bäumen (siehe Kapitel »Verzeigerte Datenstrukturen«), bei der Berechnung rekursiver Funktionen, bei der logischen Analyse von Spielen, ...

Die meisten rekursiven Unterprogramme sind relativ kompliziert und tragen damit wenig zum Verständnis der Phänomene bei, die hier eine Rolle spielen. Wir beginnen deshalb mit einer rekursiv definierten Funktion, der *Fakultät* einer ganzen Zahl $n > 0$; diese Funktion wird meist mit n! bezeichnet. n! ist definiert als $1! = 1$ und $(n+1)! = (n+1) * n!$. Jede rekursive Definition einer Funktion besteht aus mindestens einem Teil, der den Funktionswert zu bestimmten Argumenten direkt angibt (hier $1! = 1$) und einem Teil, der den Funktionswert zu anderen Argumenten durch die zu definierende Funktion selbst angibt. In unserem Beispiel könnten wir, ausgehend von $1! = 1$, sukzessive die Funktionswerte $2! = 2, 3! = 6, 4! = 24$, ... berechnen (natürlich wird n! in der Praxis als Produkt der ganzen Zahlen von 1 bis n nicht rekursiv, sondern durch eine Schleife berechnet). Ein Algorithmus zur rekursiven Berechnung von n! würde damit folgendermaßen lauten:

```
Unterprogramm          FAKULT (n, f)
        wenn           n = 1
        dann           f <– 1
        sonst          aktiviere FAKULT (n-1, g)
                       f <– n * g
Ende Unterprogramm
```

Der formale Parameter f nimmt dabei den berechneten Funktionswert auf. Mögliche Aufrufe wären:

aktiviere FAKULT (7, DE)
aktiviere FAKULT (<C>, IX)

Wir wollen nun ein Programm schreiben, das zu gegebenem Argument n im A-Register den Funktionswert n! im HL-Register berechnet; jeder Aufruf des Unterprogramms würde also von der Form

aktiviere FAKULT (<A>, HL)

sein. Beachte, daß n! sehr schnell größer wird!

```
FAKULT:  LD      HL,1      ; Funktionswert zu
                           ; Argument n = 1 laden
         CP      1         ; Argument auf 1 pruefen
         RET     Z         ; Argument ist 1,
                           ; Funktionswert 1 ist im
                           ; HL-Register
         PUSH    AF        ; Argument n wird spaeter
                           ; wieder benoetigt
         DEC     A         ; Argument fuer rekursiven
                           ; Aufruf berechnen
         CALL    FAKULT    ; (n-1)! im HL-Register
                           ; berechnen lassen
         POP     AF        ; Argument n restaurieren
         LD      B,A       ; Argument n als Zaehler benutzen
         EX      DE,HL     ; (n-1)! nach DE
         LD      HL,0      ;
MULT:    ADD     HL,DE     ; n! = n * (n-1)!
         DJNZ    MULT      ; berechnen
         RET               ; n! im HL-Register abliefern
```

Das Objekt-Programm lautet:

Adresse	Objekt-Code	Marke	Anweisung	
0000	21 01 00	FAKULT: LD	HL,1	
0003	FE 01		CP	1
0005	C8		RET	Z
0006	F5		PUSH	AF
0007	3D		DEC	A
0008	CD 00 00		CALL	FAKULT
000B	F1		POP	AF
000C	47		LD	B,A
000D	29	MULT:	ADD	HL,HL
000E	10 FD		DJNZ	MULT
0010	C9		RET	

Die wichtigste Beobachtung ist, daß wir das Argument n temporär sichern müssen. Würden wir das nicht tun, so würde nach der Rückkehr aus dem rekursiven Aufruf das A-Register einen anderen Wert enthalten, da das Unterprogramm den Wert des A-Registers verändert (Seiteneffekt!).

Wir bemerken weiter, daß jeder Aufruf von FAKULT für $n > 1$ zwei Elemente auf den Stapel wirft: die Rückkehradresse und das gesicherte Argument n. Für $n = 1$ wird nur die Rückkehradresse auf den Stapel gegeben. Diese Stapel-Elemente sammeln sich an, bis FAKULT für $n = 1$ ausgewertet ist; dann werden sie schrittweise wieder abgebaut (die Berechnung von n! hat damit eine Belastung des Stapels durch $2 * n - 1$ Elemente zur Folge). Auch dies ist eine typische Erscheinung der rekursiven Programmierung. Es gilt hier zu beachten, daß der Stapel nicht beliebig groß ist; viele geschachtelte Aufrufe von Unterprogrammen führen leicht zu einem Stapel-Überlauf.

Wir kommen nun zu einem etwas schwierigeren Problem: der Bestimmung des größten gemeinsamen Teilers ggT(a,b) zweier positiver ganzer Zahlen a und b. Bereits der Grieche Euklid kannte eine Lösung dieses Problems. Der Euklidsche Algorithmus lautet:

ggT(a,b) = a, falls a = b,
ggT(a,b) = ggT(a–b,b), falls a > b,
ggT(a,b) = ggT(a,b–a), falls a < b.

Wir setzen dies in die Formulierung eines rekursiven Unterprogramms um:

Unterprogramm ggT (a, b, g)
wenn a = b
dann g <– a
sonst **wenn** a > b
dann **aktiviere** ggT (a–b, b, g)
sonst **aktiviere** ggT (a, b–a, g)
Ende Unterprogramm

Der formale Parameter g enthält nach Ausführung des Unterprogramms den größten gemeinsamen Teiler von a und b.

Wir formulieren nun unser Programm; dabei soll das A-Register den Parameter a, das B-Register den Parameter b und wiederum das A-Register das Ergebnis g aufnehmen:

```
GGT:    CP      B       ; a und b vergleichen
        RET     Z       ; a = b, Funktionswert a
        JP      C,FALL3 ; 3. Fall liegt vor
        SUB     B       ; a <— a — b berechnen
        CALL    GGT     ; ggT(a—b,a) rekursiv aufrufen
        RET             ; Funktionswert abliefern
FALL3:  LD      C,A     ; b <— b — a berechnen
        LD      A,B
        SUB     C
        LD      B,A
        LD      A,C
        CALL    GGT     ; ggT(a,b—a) rekursiv aufrufen
        RET             ; Funktionswert abliefern
```

Warum brauchen wir hier keine Registerinhalte sichern? Die Erklärung ist einfach: Keines der verwendeten Register wird nach einem Aufruf von GGT nochmals benötigt, da ein solcher Aufruf das letzte ist, was im Unterprogramm GGT ausgeführt wird (solche Aufrufe nennt man end-rekursiv). Wenn wir uns das Unterkapitel 19.1 nochmals genau ansehen, so wird klar, daß wir das Unterprogramm GGT folgendermaßen optimieren können:

```
GGT:    CP      B       ; a und b vergleichen
        RET     Z       ; a = b, Funktionswert a
        JP      C,FALL3 ; 3. Fall liegt vor
        SUB     B       ; a <— a — b berechnen
        JP      GGT     ; ggT(a—b,a) rekursiv aufrufen
                        ; und Funktionswert abliefern
FALL3:  LD      C,A     ; b <— b — a berechnen
        LD      A,B
        SUB     C
        LD      B,A
        LD      A,C
        JP      GGT     ; ggT(a,b—a) rekursiv aufrufen
                        ; und Funktionswert abliefern
```

Wir krönen dieses Unterkapitel mit einer Perle der Programmierung: *Quicksort*, ein schnelles Sortierprogramm.

Wir wollen ein Unterprogramm QUICKSORT zur aufsteigenden Sortierung eines Felds von vorzeichenlosen ganzen 8-Bit-Zahlen (für Worte funktioniert es analog) schreiben; das Feld ist

durch zwei Zeiger z_1 und z_2 beschrieben, wobei z_1 auf das erste Feldelement zeigt, z_2 auf das letzte. Das Unterprogramm soll durch

aktiviere QUICKSORT (z_1, z_2)
aufgerufen werden.

Wenn $z_2 <= z_1$ gilt, enthält das Feld höchstens ein Element; es braucht also nicht sortiert zu werden. Wir behandeln im folgenden den Fall $z_1 < z_2$.

Nehmen wir an, das Feld sei *partitioniert*; das heißt, es gibt zwei Adressen a_1 und a_2 mit $z_1 <= a_1 <= a_2 <= z_2$ und einen Byte-Wert x (einen Schnittwert) mit den Eigenschaften

<(a)> <= x für alle Adressen a mit $z_1 <= a <= a_1$,
<(a)> = x für alle Adressen a mit $a_1 < a < a_2$,
<(a)> >= x für alle Adressen a mit $a_2 <= a <= z_2$,

Dies bedeutet, daß wir nur noch die Adreßbereiche z_1 bis a_1 und a_2 bis z_2 aufsteigend sortieren müssen, um das gesamte Feld zu sortieren. Es genügen also zwei rekursive Aufrufe

aktiviere QUICKSORT (z_1, a_1)
aktiviere QUICKSORT (a_2, z_2)

Wir führen nun zuerst eine geeignete Partitionierung des Felds herbei und rufen dann QUICKSORT zweimal rekursiv auf. Wir müssen die Partitionierung dabei so gestalten, daß $a_1 < z_2$ und $a_2 > z_1$ gilt, sonst bricht das Verfahren nicht ab; die zu sortierenden Teilbereiche des Felds müssen mit jedem Aufruf von QUICKSORT kleiner werden.

Wir werden stets ein solches x wählen, das als Wert eines Feldelements im zu sortierenden Feld vorkommt; wir hoffen, dadurch die beiden zu sortierenden Teilbereiche des Felds möglichst klein zu machen. Die Strategie des Sortierverfahrens hängt nun noch davon ab, welches Feldelement wir zur Gewinnung von x auswählen. Wir überlassen dies einem Unterprogramm WAHL, das mittels

aktiviere WAHL (z_1, z_2, x)

aufgerufen wird. Sind die Werte des Felds in willkürlicher Reihenfolge, so ist jedes Feldelement gleich gut; wir würden dann zum Beispiel

x <– <(z_1)>

wählen. Sind die Elemente des Felds schon relativ gut vorsortiert, so bietet sich an, ein Element aus der Mitte des Felds zu wählen, zum Beispiel

z <– (z_1+z_2)/2 (ganzzahliger Anteil)
x <– <(z)>

Wenn man gar keine Vermutung hat, nimmt man ein Element zufällig aus dem Feld:

z <– zufällige Adresse aus dem Bereich z_1 bis z_2
x <– <(z)>

Ein Unterprogramm für die erste Strategie würde formal lauten:

Unterprogramm WAHL (z1, z2, x)
x <– <(z_1)>
Ende Unterprogramm

Als Schnittstelle vereinbaren wir, daß z_1 im HL-Register und z_2 im DE-Register übergeben werden, und daß x im A-Register zurückgeliefert wird. Das zugehörige Programm würde damit lauten:

```
WAHL:   LD      A,(HL)      ; Schnittwert bestimmen
        RET
```

Beachte, daß das Unterprogramm WAHL keine unerwünschten Seiteneffekte aufweist.
Wir besetzen den Zeiger a_1 mit dem Wert von z_2 vor, den Zeiger a_2 mit dem Wert von z_1. Nun führen wir folgenden Prozeß durch: Wir dekrementieren a_1, solange <(a_1)> > x und $a_1 > z_1$ gilt. Ebenso inkrementieren wir a_2, solange <(a_2)> < x und $a_2 < z_2$ gilt.
Ist nach Abschluß des Verfahrens $a_2 < a_1$, so liegt ein Hindernis vor, das durch Vertauschen der Feldelemente mit den Adressen a_1 und a_2 und anschließendes Dekrementieren von a_1 und Inkrementieren von a_2 beseitigt wird. Der Prozeß wird so lange wiederholt, bis $a_1 <= a_2$ geworden ist.

Nun kann es vorkommen, daß wir durch den Prozeß $a_2 = z_1$ erhalten. In diesem Fall besitzt das Feldelement mit der Adresse z_1 den Wert x, alle übrigen Feldelemente dagegen besitzen größere Werte. Wir rufen dann

aktiviere QUICKSORT (z_1+1, z_2)

auf. Ist dagegen $a_1 = z_2$, so besitzt das Feldelement mit der Adresse z_2 den Wert x, alle anderen Feldelemente dagegen besitzen kleinere Werte. Dann rufen wir

aktiviere QUICKSORT (z_1, z_2–1)

auf. Ist schließlich $a_2 > z_1$ und $a_1 < z_2$, so rufen wir wie beabsichtigt

aktiviere QUICKSORT (z_1, a_1)
aktiviere QUICKSORT (a_2, z_2)

auf. Wir formulieren nun den Algorithmus von QUICKSORT formal:

```
Unterprogramm     QUICKSORT (z1, z2)
        wenn      <z2><=<z1>
        dann      verlasse Unterprogramm
        aktiviere WAHL (z1, z2, x)
        a1 <- <z2>
        a2 <- <z1>
        wiederhole
                  wiederhole
                            a1 <- <a1> - 1
                  solange   <(<a1>)> ><x> und <a1> ><z1>
        wiederhole
                            a2 <- <a2> + 1
                  solange   <(<a2>)><<x> und <a2><<z2>
                  wenn      <a1> ><a2>
                  dann      vertausche <(<a1>)> und <(<a2>)>
                            a1 <- <a1> - 1
                            a2 <- <a2> + 1
        bis       <a1><=<a2>
        wenn      <a2>=<z1>
        dann      z1 <- <z1> + 1
                  aktiviere QUICKSORT (z1, z2)
                  verlasse Unterprogramm
        wenn      <a1>=<z2>
        dann      z2 <- <z2> - 1
                  aktiviere QUICKSORT (z1, z2)
                  verlasse Unterprogramm
        aktiviere QUICKSORT (z1, a1)
        aktiviere QUICKSORT (a2, z2)
        verlasse Unterprogramm
Ende Unterprogramm
```

Wir benötigen mehrmals ein Unterprogramm, das zwei Adressen auf die Relation »kleiner« beziehungsweise die Relation »gleich« testet. Wir verwenden deshalb ein solches Unterprogramm, das auf die Register HL und DE angewendet wird. Das Unterprogramm schützt alle benutzten Register bis auf das F-Register. Ist der Inhalt des HL-Registers kleiner als der Inhalt des DE-Registers, so wird das Übertrag-Flag gesetzt. Stimmen beide Inhalte überein, so wird das Null-Flag gesetzt:

```
TEST:   PUSH    HL      ; benutztes Register sichern
        OR      A       ; Uebertrag-Flag ruecksetzen
        SBC     HL,DE   ; Test ausfuehren
        POP     HL      ; benutztes Register restaurieren
        RET
```

Nun können wir uns endlich an die Formulierung von QUICKSORT wagen (wieder enthält das HL-Register den Zeiger z_1, das DE-Register den Zeiger z_2). Zum besseren Verständnis des Programms habe ich an Schlüsselstellen die Belegung der Register A, HL, DE sowie des Stapels als Kommentar angegeben; dieses Verfahren ließe sich zu einem formalen Verifikationsschema ausbauen.

```
QUICKS:  PUSH    AF          ; Unterprogramm seiteneffekt-
         PUSH    BC          ; frei machen durch
         PUSH    DE          ; Sicherung aller
         PUSH    HL          ; verwendeten Register

; HL = z1, DE = z2, Stapel = z1, z2, ...

         CALL    TEST        ; auf z2 <= z1 testen
         JP      NC,FERTIG   ; z2 <= z1,
                             ; hoechstens ein Element im Feld,
                             ; nichts zu sortieren,
                             ; Ende des Unterprogramms

; HL = z1, DE = z2, Stapel = z1, z2, ...

         CALL    WAHL        ; Schnittelement bestimmen

; A = x, HL = z1, DE = z2, Stapel = z1, z2, ...

PARTIT:

; A = x, HL = a2, DE = a1, Stapel = z1, z2, ...

         EX      (SP),HL     ; z1 holen, a2 sichern
         EX      DE,HL       ; a1 zum Hauptzeiger machen

SUCHE1:

; A = x, HL = a1, DE = z1, Stapel = a2, z2, ...

         CP      (HL)        ; <x> mit <(<a1>)> vergleichen
         JP      NC,SU1END   ; <x> >= <(<a1>)>
         CALL    TEST        ; Test auf <a1> = <z1>
         JP      Z,SU1END    ; <a1> = <z1>
         DEC     HL          ; Zeiger a1 dekrementieren
         JP      SUCHE1      ; Suche fortsetzen

SU1END:
```

```
; A = x, HL = a1, DE = z1, Stapel = a2, z2, ...

        POP     BC      ; a2 temporaer holen
        EX      (SP),HL ; z2 holen, a1 sichern
        EX      DE,HL   ; z1 zugaenglich machen
        PUSH    BC      ; a2 wieder sichern
        EX      (SP),HL ; a2 holen, z1 sichern

SUCHE2:

; A = x, HL = a2, DE = z2, Stapel = z1, a1, ...

        CP      (HL)    ; auf <x><=<(<a2>)> testen
        JP      C,SU2END ; <x><<(<a2>)>
        JP      Z,SU2END ; <x>=<(<a2>)>
        CALL    TEST    ; auf <a2>=<z2> testen
        JP      Z,SU2END ; <a2>=<z2>
        INC     HL      ; Zeiger a2 inkrementieren
        JP      SUCHE2  ; Suche fortsetzen

SU2END:

; A = x, HL = a2, DE = z2, Stapel = z1, a1, ...

        POP     BC      ; z1 temporaer holen
        EX      DE,HL   ; z2 zugaenglich machen
        EX      (SP),HL ; a1 holen, z2 sichern
        PUSH    BC      ; z1 wieder sichern
        EX      DE,HL   ; a2 zum Hauptzeiger machen

; A = x, HL = a2, DE = a1, Stapel = z1, z2, ...

        CALL    TEST    ; auf <a1>><a2> testen
        JP      NC,PAREND ; <a1><=<a2>,
                        ; Partitionierung beendet
; A = x, HL = a2, DE = a1, Stapel = z1, z2, ...
        LD      B,(HL)  ; Inhalte
        EX      DE,HL   ; von
        LD      C,(HL)  ; (<a1>)
        LD      (HL),B  ; und
        EX      DE,HL   ; (<a2>)
        LD      (HL),C  ; tauschen
        DEC     DE      ; a1 dekrementieren
        INC     HL      ; a2 inkrementieren
```

```
; A = x, HL = a2, DE = a1, Stapel = z1, z2, ...

        CALL    TEST        ; auf <a1><=<a2> testen
        JP      C,PARTIT    ; <a1>><a2>,
                            ; Partitionierung fortsetzen

PAREND:

; A = x, HL = a2, DE = a1, Stapel = z1, z2, ...

        EX      DE,HL       ; a1 zugaenglich machen
        EX      (SP),HL     ; z1 holen, a1 sichern

; A = x, HL = z1, DE = a2, Stapel = a1, z2, ...

        CALL    TEST        ; auf <a2>=<z1> testen
        JP      NZ,UNGL1    ; <a2>><z1>

; A = x, HL = z1, DE = a2, Stapel = a1, z2, ...

        POP     DE          ; a2 wegwerfen, a1 holen
        POP     DE          ; a1 wegwerfen, z2 holen
        PUSH    DE          ; z2 sichern
        PUSH    HL          ; z1 sichern

; A = x, HL = z1, DE = z2, Stapel = z1, z2, ...

        INC     HL          ; QUICKSORT (z1+1, z2)
        CALL    QUICKS      ; aufrufen
        JP      FERTIG      ; Unterprogramm verlassen

UNGL1:

; A = x, HL = z1, DE = a2, Stapel = a1, z2, ...

        POP     BC          ; a1 temporaer holen
        EX      DE,HL       ; a2 zugaenglich machen
        EX      (SP),HL     ; z2 holen, a2 sichern
        EX      DE,HL       ; z1 zugaenglich machen
        PUSH    BC          ; a1 sichern
        EX      (SP),HL     ; a1 holen, z1 sichern

; A = x, HL = a1, DE = z2, Stapel = z1, a2, ...
```

```
        CALL    TEST        ; auf <a1>=<z2> testen
        JP      NZ,UNGL2    ; <a1><<z2>
```

; A = x, HL = a_1, DE = z_2, Stapel = z_1, a_2, ...

```
        POP     HL          ; a1 wegwerfen, z1 holen
        POP     BC          ; a2 wegwerfen
        PUSH    DE          ; z2 sichern
        PUSH    HL          ; z1 sichern
```

; A = x, HL = z_1, DE = z_2, Stapel = z_1, z_2, ...

```
        DEC     DE          ; QUICKSORT (z1, z2-1)
        CALL    QUICKS      ; aufrufen
        JP      FERTIG      ; Unterprogramm verlassen
```

UNGL2:

; A = x, HL = a_1, DE = z_2, Stapel = z_1, a_2, ...

```
        EX      DE,HL       ; z2 zugaenglich machen
        EX      (SP),HL     ; z1 holen, z2 sichern
```

; A = x, HL = z_1, DE = a_1, Stapel = z_2, a_2, ...

```
        CALL    QUICKS      ; QUICKSORT (z1, a1) aufrufen
```

; A = x, HL = z_1, DE = a_1, Stapel = z_2, a_2, ...

```
        EX      (SP),HL     ; z2 holen, z1 sichern
        EX      DE,HL       ; a1 zugaenglich machen
        POP     HL          ; a1 wegwerfen, z1 holen
        POP     BC          ; a2 temporaer holen
        PUSH    DE          ; z2 sichern
        PUSH    BC          ; a2 sichern
        EX      (SP),HL     ; a2 holen, z1 sichern
```

; A = x, HL = a_2, DE = z_2, Stapel = z_1, z_2, ...

```
        CALL    QUICKS      ; QUICKSORT (a2, z2) aufrufen
```

FERTIG:

```
; A = x, Stapel = z1, z2, ...

; Restaurieren aller Register und Verlassen des Unterprogramms

        POP     HL      ; alle
        POP     DE      ; gesicherten
        POP     BC      ; Register
        POP     AF      ; restaurieren
        RET             ; Ende von QUICKSORT
```

Mit dem Programm können auch Felder von Zeichen aufsteigend lexikalisch sortiert werden. Da QUICKSORT einen erheblichen Teil seiner Aktivitäten auf organisatorische Funktionen verwendet, zeigt sich die Überlegenheit des Verfahrens über einfachere Verfahren erst bei größeren Datenmengen.

Neben den rekursiven Unterprogrammen gibt es auch sogenannte *verschränkt rekursive* Unterprogramme. Zwei Unterprogramme A und B sind verschränkt rekursiv, wenn A das Unterprogramm B aufruft und B wiederum das Unterprogramm A aufruft; sieht man die Wirkung von A und B zusammen als die Wirkung eines Unterprogramms C an, so wäre C ein rekursives Unterprogramm. Es können auch mehr als zwei Unterprogramme verschränkt rekursiv sein. Ein Beispiel für zwei verschränkt rekursive Unterprogramme:

Ein Unterprogramm A zur Behandlung von Fehlern in Ein-/Ausgabe-Unterprogrammen ruft ein Unterprogramm B auf, das eine Fehlermeldung ausgibt; kommt es dabei erneut zu einem Fehler, so ruft das Unterprogramm B zur Fehlerbehandlung wiederum das Unterprogramm A auf, das im Gegenzug wieder eine Meldung durch Aufruf des Unterprogramms B ausgibt. Dieser Prozeß kann sogar zu einer nicht endenden Rekursion führen (das System »hängt sich auf«).

Übungen

1. Ein arithmetischer Ausdruck, der aus vorzeichenlosen ganzen Zahlen, Klammern und den beiden Operationssymbolen »+« und »-« zusammengesetzt ist, kann eine der folgenden Formen haben:
   - Der Ausdruck ist eine Zahl: ausdruck = zahl
   - Der Ausdruck ist eine Summe: ausdruck = ausdruck_1 + ausdruck_2
   - Der Ausdruck ist eine Differenz: ausdruck = ausdruck_1 - ausdruck_2
   - Der Ausdruck ist ein Klammerausdruck: ausdruck = (ausdruck_1)

   Diese Definition eines arithmetischen Ausdrucks ist rekursiv aufgebaut. Zur Behandlung solcher Ausdrücke eignen sich deshalb in besonderem Maße rekursive Programme.
   Schreibe ein Programm, das prüft, ob durch eine Zeichenkette ein arithmetischer Ausdruck dargestellt wird; verwende dabei als Zahlbereich die Zahlen 0, 1, ..., 9 und stelle diese durch jeweils eine ASCII-codierte Dezimalziffer dar.

19.9 Eintritts-invariante Unterprogramme

Während die Anforderungen an die Sicherung von Daten bei rekursiven Unterprogrammen im wesentlichen davon abhängen, ob die betreffenden Daten nach Ausführung eines rekursiven Aufrufs noch zur Verfügung stehen müssen, sind in einem System, in dem sich Programme gegenseitig unterbrechen können (siehe Kapitel »Unterbrechungen«) härtere Bedingungen an die Struktur von Unterprogrammen zu stellen; dort verlangt man eintritts-invariante (engl. reentrant) Unterprogramme. Ein eintritts-invariantes Unterprogramm schafft sich für jeden Aufruf einen eigenen Speicherbereich, in dem seine Daten untergebracht sind; zu diesem Speicherbereich hat diese Inkarnation des Unterprogramms den alleinigen Zugriff.

Die Compiler von Sprachen wie PASCAL lösen das Problem folgendermaßen: Jedes Unterprogramm reserviert sich bei jedem Aufruf auf dem Stapel Speicherplatz für seine Variablen. Der reservierte Speicherplatz liegt oberhalb der Rückkehradresse, die sich beim Betreten des Unterprogramms als oberstes Element auf dem Stapel befindet. Die Adressierung der Variablen erfolgt indirekt, entweder über Indexregister oder über das HL-Register. Vor dem Verlassen des Unterprogramms wird der Stapel-Zeiger wieder auf seinen alten Wert gesetzt. Wir demonstrieren die Methode an folgendem Beispiel:

Bei der Verarbeitung von Zeichenketten verwenden wir Deskriptoren, die aus einer Längenangabe (ein Byte) und der Adresse des Textes bestehen. Wir wollen ein Unterprogramm STERNE schreiben, das auf dem Stapel den Deskriptor einer Zeichenkette als Parameter erhält; der Parameterblock besteht aus drei aufeinanderfolgenden Bytes, wobei das Byte mit der niedrigsten Adresse die Längenangabe enthält. Das Unterprogramm STERNE soll nun den Deskriptor der ersten Teil-Zeichenkette der übergebenen Zeichenkette bereitstellen, die vorne und hinten von einem Stern »*« begrenzt wird; die Sterne gehören nicht zur Teil-Zeichenkette. Ist in der übergebenen Zeichenkette eine solche Teil-Zeichenkette enthalten, so soll der Deskriptor der Teil-Zeichenkette einem Ausgabe-Unterprogramm AUSGAB als Parameter auf dem Stapel übergeben werden; andernfalls hat STERNE keine Wirkung.

Wir reservieren uns als erstes auf dem Stapel drei Bytes Speicherplatz für den Deskriptor der Teil-Zeichenkette. Dieser Deskriptor stellt gleichzeitig den Parameterblock für das Unterprogramm AUSGAB dar; die Form des Parameterblocks von AUSGAB ist dieselbe wie die des Parameterblocks von STERNE. Den neuen Wert des Stapel-Zeigers verwenden wir dann als Basis-Adresse eines Verbunds (auf dem Stapel), der aus dem Deskriptor der Teil-Zeichenkette, der Rückkehradresse für STERNE und dem Deskriptor der Zeichenkette besteht:

```
STERNE:  DEC   SP          ; Platz fuer Variablen
         DEC   SP          ; auf dem Stapel
         DEC   SP          ; reservieren
         LD    IX,0        ; Zeiger auf den
         ADD   IX,SP       ; Datenblock berechnen
         LD    C,(IX+5)    ; Laenge der Zeichenkette holen
         LD    B,0         ; und zu 16-Bit-Groesse machen
         LD    L,(IX+6)    ; Adresse des Texts der
```

```
        LD      H,(IX+7)        ; Zeichenkette holen
        LD      A,'*'           ; Begrenzungszeichen laden
        CPIR                    ; in Zeichenkette nach
                                ; Begrenzungszeichen suchen
        JP      PO,FERTIG       ; Zeichenkette zu Ende,
                                ; Begrenzungszeichen hoechstens
                                ; einmal in Zeichenkette
                                ; enthalten, nichts zu tun
        LD      (IX+1),L        ; Adresse des Texts der
        LD      (IX+2),H        ; Teil-Zeichenkette abspeichern
        LD      E,0             ; Zaehler fuer Laenge der
                                ; Teil-Zeichenkette aufsetzen
SUCHE:  CPI                     ; ein Zeichen der Zeichenkette
                                ; absuchen
        JP      Z,GEFUND        ; zweiter Stern gefunden
        P       PO,FERTIG       ; Zeichenkette zu Ende,
                                ; Begrenzungszeichen hoechstens
                                ; einmal in Zeichenkette
                                ; enthalten, nichts zu tun
        INC     E               ; abgesuchtes Zeichen gehoert zur
                                ; Teil-Zeichenkette, mitzaehlen
        JP      SUCHE           ; Rest der Zeichenkette absuchen
GEFUND: LD      (IX+0),E        ; Laenge der Teil-Zeichenkette
                                ; abspeichern
        CALL    AUSGAB          ; Deskriptor uebergeben
FERTIG: INC     SP              ; Stapel-Zeiger
        INC     SP              ; auf alten Wert
        INC     SP              ; setzen
        RET                     ; STERNE verlassen
```

Beim Z80 kann nun nicht direkt auf Variablen gearbeitet werden; alle Daten-Operationen benötigen Register. Die Register soll sich ein Unterprogramm natürlich nicht für den eigenen Gebrauch reservieren; sie stehen allen Unterprogrammen zur Verfügung. Deshalb lassen sich eintritts-invariante Unterprogramme im strengen Sinn mit dem Z80 gar nicht realisieren. Wir können aber eine zusammengehörige Menge von Unterprogrammen so schreiben, daß sich diese bei gegenseitigen Aufrufen und Unterbrechungen (dieses Teil verschieben wir auf das Kapitel »Unterbrechungen«) so verhalten, als wären sie eintritts-invariant. Es reicht in diesem Fall nämlich aus, wenn unmittelbar nach dem Betreten eines Unterprogramms alle Register (mit Ausnahme des Stapel-Zeigers und des Befehls-Zählers), die durch dieses Unterprogramm (möglicherweise) verändert werden, auf dem Stapel gesichert und vor Verlassen des Unterprogramm restauriert werden; allerdings müssen alle beteiligten Unterprogramme so geschrieben sein, damit sie gegenseitig eintritts-invariant erscheinen. Für unser Beispiel würde dies folgende Ergänzungen erforderlich machen:

```
STERNE:  PUSH   AF          ; alle
         PUSH   BC          ; verwendeten
         PUSH   DE          ; Register
         PUSH   HL          ; zu Beginn
         PUSH   IX          ; sichern
         DEC    SP          ; Platz fuer Variablen
         DEC    SP          ; auf dem Stapel
         DEC    SP          ; reservieren
         LD     IX,0        ; Zeiger auf den
         ADD    IX,SP       ; Datenblock berechnen
         LD     C,(IX+15)   ; Laenge der Zeichenkette holen
         LD     B,0         ; und zu 16-Bit-Groesse machen
         LD     L,(IX+16)   ; Adresse des Texts der
         LD     H,(IX+17)   ; Zeichenkette holen
         LD     A,'*'       ; Begrenzungszeichen laden
         CPIR               ; in Zeichenkette nach
                            ; Begrenzungszeichen suchen
         JP     PO,FERTIG   ; Zeichenkette zu Ende,
                            ; Begrenzungszeichen hoechstens
                            ; einmal in Zeichenkette
                            ; enthalten, nichts zu tun
         LD     (IX+1),L    ; Adresse des Texts der
         LD     (IX+2),H    ; Teil-Zeichenkette abspeichern
         LD     E,0         ; Zaehler fuer Laenge der
                            ; Teil-Zeichenkette aufsetzen
SUCHE:   CPI                ; ein Zeichen der Zeichenkette
                            ; absuchen
         JP     Z,GEFUND    ; zweiter Stern gefunden
         JP     PO,FERTIG   ; Zeichenkette zu Ende,
                            ; Begrenzungszeichen hoechstens
                            ; einmal in Zeichenkette
                            ; enthalten, nichts zu tun
         INC    E           ; abgesuchtes Zeichen gehoert zur
                            ; Teil-Zeichenkette, mitzaehlen
         JP     SUCHE       ; Rest der Zeichenkette absuchen
GEFUND:  LD     (IX+0),E    ; Laenge der Teil-Zeichenkette
                            ; abspeichern
         CALL   AUSGAB      ; Deskriptor uebergeben
FERTIG:  INC    SP          ; Stapel-Zeiger
         INC    SP          ; auf alten Wert
         INC    SP          ; setzen
         POP    IX          ; alte
         POP    HL          ; Werte
```

```
POP     DE      ; der
POP     BC      ; Register
POP     AF      ; wiederherstellen
RET             ; STERNE verlassen
```

Auch wenn es nicht zwingend notwendig ist, alle Programme eintritts-invariant zu schreiben, so ist es doch eine sichere Technik für modulares Programmieren; Voraussetzung für ein durchgängiges Anwenden dieser Methode ist jedoch, daß alle Ergebnisse über den Stapel zurückgegeben werden.

Übungen

1. Schreibe ein eintritts-invariantes Programm, dem zwei ganze Zahlen als 16-Bit-Größen in 2-Komplement-Darstellung auf dem Stapel übergeben werden, das die Summe und die Differenz der beiden Zahlen berechnet und auf dem Stapel für ein weiteres Unterprogramm zur Verfügung stellt.

20
Puffer

Puffer dienen der Kommunikation zwischen einem Daten-Produzenten und einem Daten-Konsumenten. Als Produzent kommt zum Beispiel ein Unterprogramm in Frage, das ständig die Tastatur beobachtet und von ihr gegebenenfalls Zeichen einliest; die Zeichen kommen dann in den Puffer, wo sie darauf warten, daß der Konsument - ein anderes Unterprogramm - sie abholt und verarbeitet. Bei diesem Mechanismus soll die Reihenfolge der erzeugten Daten mit der Reihenfolge der verbrauchten Daten übereinstimmen. Puffer heißen deshalb auch *Warteschlangen* oder kurz *Schlangen* (engl. queue; wer kennt nicht das queueing der Engländer an der Bushaltestelle?). Als Abkürzung in der englischsprachigen Literatur hat sich die Bezeichnung *FIFO* (first in, first out) durchgesetzt.

Puffer treten in verschiedenen Formen auf. Die einfachste Form ist der *Blockpuffer*. Der Produzent füllt den Blockpuffer von vorne, bis kein Platz mehr ist; dann wartet er darauf, bis der Konsument den Puffer völlig entleert hat. Der Konsument leert den Puffer auch von vorne; wenn er alle Zeichen, die der Produzent bisher in den Puffer geschrieben hat, abgeholt hat, muß er warten, bis wieder neue Zeichen vom Produzenten angekommen sind. Ist der Puffer völlig entleert, so wird er wieder von vorne gefüllt.

Wenn man nicht möchte, daß Produzent und Konsument gleichzeitig auf demselben Puffer arbeiten, geht man zu *Wechselpuffern* über. Dabei verwendet man mehrere (meist zwei) Blockpuffer. Der Produzent schreibt erst einen ganzen Puffer voll und übergibt ihn dann dem Konsumenten; dieser leert ihn völlig und gibt ihm dem Produzenten zurück. Während der Konsument einen bestimmten Puffer leert, füllt der Produzent einen anderen Puffer. So arbeitet zu jedem bestimmten Zeitpunkt jeder auf seinem eigenen Puffer.

Eine raffinierte Form von Puffern sind die *Ringpuffer*. Ein Ringpuffer ist ein Blockpuffer, den der Produzent sofort wieder von vorne füllt, wenn er am Ende des Puffers angelangt ist und wenn der Konsument überhaupt schon ein Element aus dem Puffer entfernt hat. Man kann sich einen Ringpuffer als ringförmig geschlossenen Blockpuffer vorstellen.

Puffer können auch durch Listen realisiert werden (siehe Kapitel »Verzeigerte Datenstruk-

turen«); dies empfiehlt sich besonders, wenn die Elemente des Puffers verschiedene Längen haben. Desgleichen kommt eine Darstellung als Tabelle in Frage (siehe Kapitel »Tabellen«).

20.1 Blockpuffer

Im folgenden wollen wir stets Puffer behandeln, deren Elemente Zeichen sind.

Wir stellen einen Blockpuffer durch einen zusammenhängenden Speicherbereich mit Anfangsadresse ANFANG und Endadresse ENDE dar. Ein Beispiel für einen Blockpuffer mit 256 Elementen würde lauten:

```
PUFFL:    EQU     256              ; Laenge des Puffers
ENDE:     EQU     ANFANG+PUFFL-1   ; Endadresse des Puffers
ANFANG:   DEFS    PUFFL            ; Speicherplatz fuer Puffer
```

Es sind vier Operationen auf dem Puffer nötig:

Für den Produzenten ist wichtig zu wissen, ob der Puffer voll ist (Operation VOLL); in einem nicht gänzlich gefüllten Puffer kann er ein Zeichen ablegen (Operation FUELLE).

Für den Konsumenten ist interessant, ob der Puffer leer ist (Operation LEER); aus einem nicht leeren Puffer kann er ein Zeichen entnehmen (Operation LEERE).

Wir legen nun für den Produzenten und den Konsumenten je einen Puffer-Zeiger an. Der Zeiger PZEIG des Produzenten zeigt auf den nächsten freien Speicherplatz im Puffer; zu Beginn hat PZEIG also den Wert ANFANG. Ist der Puffer voll, so hat PZEIG den Wert ENDE+1. Der Zeiger KZEIG des Konsumenten zeigt auf das nächste zu holende Zeichen im Puffer; zu Beginn besitzt also auch KZEIG den Wert ANFANG. Der Puffer ist leer, wenn KZEIG und PZEIG den gleichen Wert besitzen; wir setzen in diesem Fall KZEIG und PZEIG stets auf den Wert ANFANG zurück.

Unter Hinzunahme der beiden Puffer-Zeiger lautet nun die Vereinbarung eines Puffers:

```
PUFFL:    EQU     256              ; Laenge des Puffers
ENDE:     EQU     ANFANG+PUFFL-1   ; Endadresse des Puffers
PZEIG:    DEFS    2                ; Puffer-Zeiger des Produzenten
KZEIG:    DEFS    2                ; Puffer-Zeiger des Konsumenten
ANFANG:   DEFS    PUFFL            ; Speicherplatz fuer Puffer
```

Wir beschreiben nun die Pufferoperationen abstrakt; zu den anfänglich genannten vier Operationen kommt noch die Initialisierung der Puffer-Zeiger hinzu:

Unterprogramm INIT
PZEIG <– ANFANG
KZEIG <– ANFANG
Ende Unterprogramm

```
Unterprogramm          VOLL (w)
        wenn           <PZEIG> = ENDE+1
        dann           w <– wahr
        sonst          w <– falsch
Ende Unterprogramm

Unterprogramm          LEER (w)
        wenn           <PZEIG> = ANFANG
        dann           w <– wahr
        sonst          w <– falsch
Ende Unterprogramm

Unterprogramm          FUELLE (x)
        aktiviere VOLL (w)
        wenn           <w> = wahr
        dann           Fehlermeldung
        sonst          (<PZEIG>) <– <x>
                       PZEIG <– <PZEIG> + 1
Ende Unterprogramm

Unterprogramm          LEERE (x)
        aktiviere LEER (w)
        wenn           <w> = wahr
        dann           Fehlermeldung
        sonst          x <– <(<KZEIG>)>
                       KZEIG <– <KZEIG> + 1
                       wenn    <KZEIG> = <PZEIG>
                       dann    aktiviere INIT
Ende Unterprogramm
```

Diese Beschreibung setzen wir nun in offensichtlicher Weise in Unterprogramme um. Alle Unterprogramme schützen die benutzten Register, mit Ausnahme der Register zur Ergebnisrückgabe. Wir beginnen mit dem Unterprogramm INIT:

```
INIT:   PUSH    HL          ; Registerinhalt sichern
        LD      HL,ANFANG   ; Initialwert laden
        LD      (PZEIG),HL  ; Zeiger initialisieren
        LD      (KZEIG),HL  ; Zeiger initialisieren
        POP     HL          ; Register restaurieren
        RET
```

Als Parameter-Register für die Unterprogramme VOLL und LEER wählen wir das Null-Flag; gesetztes Null-Flag steht dabei für »wahr«, gelöschtes Null-Flag für »falsch«:

```
VOLL:   PUSH    HL          ; Registerinhalte
        PUSH    DE          ; sichern
        LD      HL,(PZEIG)  ; Produzenten-Zeiger holen
        LD      DE,ENDE     ; Testgroesse =
        SCF                 ; ENDE + 1
        SBC     HL,DE       ; auf vollen Puffer testen
        POP     DE          ; Register
        POP     HL          ; restaurieren
        RET
LEER:   PUSH    HL          ; Registerinhalte
        PUSH    DE          ; sichern
        LD      HL,(PZEIG)  ; Produzenten-Zeiger holen
        LD      DE,ANFANG   ; Testgroesse =
        OR      A           ; ANFANG
        SBC     HL,DE       ; auf leeren Puffer testen
        POP     DE          ; Register
        POP     HL          ; restaurieren
        RET
```

Nun kommt das Unterprogramm FUELLE an die Reihe. Das Zeichen wollen wir dabei im A-Register übergeben; bei Fehler wollen wir das Null-Flag setzen, sonst aber keine Operation durchführen:

```
FUELLE: CALL    VOLL        ; pruefen, ob Puffer voll
        RET     Z           ; Puffer voll, Fehler
        PUSH    HL          ; Registerinhalt sichern
        LD      HL,(PZEIG)  ; Produzenten-Zeiger holen
        LD      (HL),A      ; Zeichen ablegen
        INC     HL          ; auf naechstes Zeichen zeigen
        LD      (PZEIG),HL  ; Produzenten-Zeiger sichern
        POP     HL          ; Register restaurieren
        RET
```

Zuletzt noch das Unterprogramm LEERE, welches das Zeichen im A-Register zurückliefert und bei einem Fehler das Null-Flag setzt:

```
LEERE:  CALL    LEER        ; pruefen, ob Puffer leer
        RET     Z           ; Puffer leer, Fehler
        PUSH    HL          ; Registerinhalte
        PUSH    DE          ; sichern
        LD      HL,(KZEIG)  ; Konsumenten-Zeiger holen
        LD      A,(HL)      ; Zeichen aus Puffer nehmen
        PUSH    AF          ; und sichern
```

```
        INC     HL              ; auf naechstes Zeichen zeigen
        LD      (KZEIG),HL      ; Konsumenten-Zeiger sichern
        LD      DE,(PZEIG)      ; Produzenten-Zeiger holen
        OR      A               ; pruefen, ob Puffer-Zeiger
        SBC     HL,DE           ; uebereinstimmen
        CALL    Z,INIT          ; Puffer leer,
                                ; Puffer-Zeiger ruecksetzen
        POP     AF              ; alle
        POP     DE              ; Register
        POP     HL              ; restaurieren
        RET
```

Die Unterprogramme bilden keine Menge gegenseitig eintritts-invarianter Programme, da sie auf gemeinsamen Variablen, den Puffer-Zeigern, arbeiten. Wir werden aber im Unterkapitel »Unterbrechungen und Puffer-Bearbeitung« eine Methode kennenlernen, durch Modifikation der Unterprogramme quasi eintritts-invariante Programme zu erzwingen.

Übungen

1. Die vorgestellten Programme zur Puffer-Bearbeitung sind weder in bezug auf die benötigte Rechenzeit noch in bezug auf die Länge des Objekt-Codes optimal; versuche sie zu optimieren, ohne die äußeren Bedingungen zu verändern.

20.2 Wechselpuffer

Bei Verwendung eines Blockpuffers können sich Produzent und Konsument gegenseitig stören. Um Konsument und Produzent nahezu zu entkoppeln, kann man mehrere Puffer verwenden. Der Produzent schreibt dann zuerst einen Puffer voll, übergibt ihn dem Konsumenten und beginnt den nächsten Puffer zu füllen. Der Konsument leert die ihm übergebenen Puffer vollständig und gibt sie dem Produzenten zurück. Wir betrachten ein Beispiel mit zwei Wechselpuffern:

Wie beim einfachen Blockpuffer gibt es wieder einen Produzenten-Zeiger PZEIG und einen Konsumenten-Zeiger KZEIG; beide zeigen jedoch bei Wechselpuffern auf verschiedene Puffer. Wir nehmen zunächst folgende Speicherstruktur an:

```
PUFFL:  EQU     256             ; Laenge der Puffer
ENDE1:  EQU     ANF1+PUFFL-1    ; Endadresse des ersten Puffers
ENDE2:  EQU     ANF2+PUFFL-1    ; Endadresse des zweiten Puffers
PZEIG:  DEFS    2               ; Produzenten-Zeiger
KZEIG:  DEFS    2               ; Konsumenten-Zeiger
ANF1:   DEFS    PUFFL           ; erster Puffer
ANF2:   DEFS    PUFFL           ; zweiter Puffer
```

Zunächst sind beide Puffer leer. Wir weisen deshalb dem Produzenten den ersten Puffer zu. Der zweite Puffer wird dem Konsumenten zugewiesen; er ist ebenfalls leer. Wir können dies so interpretieren, daß der Konsument seinen Puffer bereits geleert hat und nun auf Zuweisung eines neuen Puffers wartet. Um diesen Zustand zu kennzeichnen, lassen wir den Zeiger des Konsumenten hinter den zweiten Puffer zeigen. Die Initialisierungssequenz lautet also:

PZEIG <– ANF1
KZEIG <– ENDE2+1

Hat der Produzent seinen ersten Puffer gefüllt, ist also <PZEIG> = ENDE1+1 geworden, so werden die beiden Puffer getauscht, natürlich nur in Form der entsprechenden Zeiger:

PZEIG <– ANF2
KZEIG <– ANF1

Dieser Tausch passiert immer dann, wenn der eine Puffer ganz voll, der andere ganz leer ist, wenn also <PZEIG> = ENDE1+1 und <KZEIG> = ENDE2+1 ist. Nun wiederholt sich dasselbe mit vertauschten Puffern. Irgendwann hat der Produzent seinen Puffer wieder gefüllt, der Konsument seinen Puffer geleert; es gilt dann <PZEIG> = ENDE2+1 und <KZEIG> = ENDE1+1. Nun wird wieder zurückgetauscht:

PZEIG <– ANF1
KZEIG <– ANF2

Da wir nicht wissen, ob der Produzent zuerst seinen Puffer gefüllt oder der Konsument seinen Puffer geleert hat, müssen wir sowohl nach vollständiger Füllung des Produzenten-Puffers als auch nach vollständiger Leerung des Konsumenten-Puffers prüfen, ob ein Tausch der Puffer erforderlich ist.

Damit wir einfach prüfen können, ob ein Zeiger hinter das Ende seines Puffers weist, speichern wir diese Adressen als Testwerte zusammen mit den Puffer-Zeigern ab; wir modifizieren also unsere Speicherstruktur folgendermaßen:

```
PUFFL:    EQU     256             ; Laenge der Puffer
ENDE1:    EQU     ANF1+PUFFL-1    ; Endadresse des ersten Puffers
ENDE2:    EQU     ANF2+PUFFL-1    ; Endadresse des zweiten Puffers
PZEIG:    DEFS    2               ; Produzenten-Zeiger
PENDE:    DEFS    2               ; Endwert des Produzenten-Zeigers
KZEIG:    DEFS    2               ; Konsumenten-Zeiger
KENDE:    DEFS    2               ; Endwert des Konsumenten-Zeigers
ANF1:     DEFS    PUFFL           ; erster Puffer
ANF2:     DEFS    PUFFL           ; zweiter Puffer
```

Beim Tausch der Puffer werden die Endwerte getauscht; die jeweilige Anfangsadresse ist der

Endwert minus der Pufferlänge. Wir können deshalb die Operationen auf den Puffern folgendermaßen formulieren:

```
Unterprogramm       INIT
                    PZEIG <– ANF1
                    PENDE <– ENDE1+1
                    KZEIG <– ENDE2+1
                    KENDE <– ENDE2+1
Ende Unterprogramm

Unterprogramm       VOLL (w)
        wenn        <PZEIG> = <PENDE>
        dann        w <– wahr
        sonst       w <– falsch
Ende Unterprogramm

Unterprogramm       LEER (w)
        wenn        <KZEIG> = <KENDE>
        dann        w <– wahr
        sonst       w <– falsch
Ende Unterprogramm

Unterprogramm       TAUSCH
        PENDE & KENDE <– <KENDE> & <PENDE>
        PZEIG <– <PENDE> – PUFFL
        KZEIG <– <KENDE> – PUFFL
Ende Unterprogramm

Unterprogramm       FUELLE (x)
        aktiviere VOLL (w)
        wenn        <w> = wahr
        dann        Fehlermeldung
        sonst       (<PZEIG>) <– <x>
                    PZEIG <– <PZEIG> + 1
                    aktiviere VOLL (w)
                    wenn        <w> = wahr
                    dann        aktiviere LEER (w)
                                wenn        <w> = wahr
                                dann        aktiviere TAUSCH
Ende Unterprogramm

Unterprogramm       LEERE (x)
        aktiviere LEER (w)
```

```
wenn    <w> = wahr
dann    Fehlermeldung
sonst   x <– <(<KZEIG>)>
        KZEIG <– <KZEIG> + 1
        aktiviere LEER (w)
        wenn    <w> = wahr
        dann    aktiviere VOLL (w)
                wenn    <w> = wahr
                dann    aktiviere TAUSCH
```

Ende Unterprogramm

Nun schreiben wir wieder Unterprogramme ohne unerwünschte Seiteneffekte. Wie bisher dient das A-Register zum Transport eines Elements in einen oder aus einem Puffer; das Null-Flag zeigt wieder einen vollen Puffer, leeren Puffer oder Fehler an:

```
INIT:   PUSH    HL          ; Registerinhalt sichern
        LD      HL,ANF1     ; Anfangsadresse des
                            ; ersten Puffers
        LD      (PZEIG),HL  ; Produzenten-Zeiger
                            ; initialisieren
        LD      HL,ENDE1+1  ; Endwert des Puffer-Zeigers
                            ; fuer ersten Puffer
        LD      (PENDE),HL  ; Endwert des Produzenten-Zeigers
                            ; initialisieren
        LD      HL,ENDE2+1  ; Endwert des Puffer-Zeigers
                            ; fuer zweiten Puffer
        LD      (KZEIG),HL  ; Konsumenten-Zeiger
                            ; initialisieren
        LD      (KENDE),HL  ; Endwert des Konsumenten-Zeigers
                            ; initialisieren
        POP     HL          ; Register restaurieren
        RET

VOLL:   PUSH    HL          ; Registerinhalte
        PUSH    DE          ; sichern
        LD      HL,(PZEIG)  ; Produzenten-Zeiger holen
        LD      DE,(PENDE)  ; mit Endwert
        R       A           ; fuer Produzenten-Zeiger
        SBC     HL,DE       ; vergleichen
        POP     DE          ; Register
        POP     HL          ; restaurieren
        RET
```

```
LEER:   PUSH    HL          ; Registerinhalte
        PUSH    DE          ; sichern
        LD      HL,(KZEIG)  ; Konsumenten-Zeiger holen
        LD      DE,(KENDE)  ; mit Endwert
        OR      A           ; fuer Konsumenten-Zeiger
        SBC     HL,DE       ; vergleichen
        POP     DE          ; Register
        POP     HL          ; restaurieren
        RET

TAUSCH: PUSH    HL          ; Register-
        PUSH    DE          ; inhalte
        PUSH    BC          ; sichern
        LD      DE,(PENDE)  ; Endwert des Produzenten-Zeigers
        LD      HL,(KENDE)  ; Endwert des Konsumenten-Zeigers
        LD      BC,PUFFL    ; Laenge der Puffer laden
        LD      (PENDE),HL  ; Endwerte fuer
        LD      (KENDE),DE  ; Puffer-Zeiger vertauschen
        OR      A           ; Anfangsadresse des
        SBC     HL,BC       ; Produzenten-Puffers berechnen
        LD      (PZEIG),HL  ; und in Produzenten-Zeiger laden
        EX      DE,HL       ; Puffer-Zeiger tauschen
        OR      A           ; Anfangsadresse des
        SBC     HL,BC       ; Konsumenten-Puffers berechnen
        LD      (KZEIG),HL  ; und in Konsumenten-Zeiger laden
        POP     BC          ; alle
        POP     DE          ; Register
        POP     HL          ; restaurieren
        RET

FUELLE: CALL    VOLL        ; pruefen, ob Produzenten-Puffer
                            ; voll ist
        RET     Z           ; Produzenten-Puffer voll, Fehler
        PUSH    HL          ; Registerinhalte
        PUSH    AF          ; sichern
        LD      HL,(PZEIG)  ; Produzenten-Zeiger holen
        LD      (HL),A      ; Zeichen im Puffer ablegen
        INC     HL          ; auf naechstes Zeichen zeigen
        LD      (PZEIG),HL  ; Produzenten-Zeiger abspeichern
        CALL    VOLL        ; pruefen, ob Produzenten-Puffer
                            ; jetzt voll ist
        CALL    Z,LEER      ; eventuell pruefen, ob
                            ; Konsumenten-Puffer leer ist
```

```
                CALL    Z,TAUSCH    ; wenn Produzenten-Puffer voll und
                                    ; Konsumenten-Puffer leer,
                                    ; Puffer tauschen
                POP     AF          ; Register
                POP     HL          ; restaurieren
                RET

LEERE:          CALL    LEER        ; pruefen, ob Konsumenten-Puffer
                                    ; leer ist
                RET     Z           ; Konsumenten-Puffer leer, Fehler
                PUSH    HL          ; Registerinhalt sichern
                LD      HL,(KZEIG)  ; Konsumenten-Zeiger holen
                LD      A,(HL)      ; Zeichen aus Puffer entnehmen
                PUSH    AF          ; und sichern
                INC     HL          ; auf naechstes Zeichen zeigen
                LD      (KZEIG),HL  ; Konsumenten-Zeiger abspeichern
                CALL    LEER        ; pruefen, ob Konsumenten-Puffer
                                    ; jetzt leer ist
                CALL    Z,VOLL      ; eventuell pruefen, ob
                                    ; Produzenten-Puffer voll ist
                CALL    Z,TAUSCH    ; wenn Produzenten-Puffer voll und
                                    ; Konsumenten-Puffer leer,
                                    ; Puffer tauschen
                POP     AF          ; Register
                POP     HL          ; restaurieren
                RET
```

Überlegen Sie, wie die beiden aufeinanderfolgenden bedingten Unterprogramm-Aufrufe zusammenwirken!

Übungen

1. Optimiere die Unterprogramme zur Wechselpuffer-Bearbeitung.

20.3 Ringpuffer

Sowohl Blockpuffer als auch Wechselpuffer haben den Nachteil, daß möglicherweise der Produzent oder der Konsument warten muß, obwohl ein Teil des Puffers eigentlich zur Verfügung steht; beim Blockpuffer betrifft dies allerdings nur den Produzenten. Ringpuffer vermeiden diesen Nachteil; sie erlauben dem Produzenten, so lange zu schreiben, bis der Puffer völlig gefüllt ist, und dem Konsumenten, alle Zeichen zu lesen, die der Produzent bisher geschrieben hat.

Die Vorgehensweise ist zunächst wie beim Blockpuffer. Erreicht allerdings der Produzent das Ende des Puffers, so darf er sofort wieder den bisher geleerten Teil des Puffers beschreiben. Man kann sich vorstellen, daß der Puffer zum Ring geschlossen ist und auf das letzte Byte des unterlegten Blockpuffers wieder das erste Byte folgt.

Da bei diesem Verfahren der Produzenten-Zeiger auch vor dem Konsumenten-Zeiger stehen kann, reichen die Zeiger alleine nicht aus, um festzustellen, ob der Puffer voll beziehungsweise leer ist; in beiden Fällen stimmen nämlich die Zeiger überein. Wir wissen jedoch, daß der Puffer nach einer Lese-Operation nicht voll sein kann, nach einer Schreib-Operation nicht leer; zu Beginn ist der Puffer leer.

Wir legen uns deshalb zwei Flags an, die anzeigen, ob der Puffer voll oder leer ist (oder keines von beiden) und aktualisieren diese Flags laufend. Die Speicherstruktur könnte damit folgendermaßen aussehen:

```
PUFFL:    EQU     256              ; Laenge des Puffers
ENDE:     EQU     ANFANG+PUFFL-1   ; Endadresse des Puffers
FLAGS:    DEFS    1                ; Voll-Flag und Leer-Flag
PZEIG:    DEFS    2                ; Puffer-Zeiger des Produzenten
KZEIG:    DEFS    2                ; Puffer-Zeiger des Konsumenten
ANFANG:   DEFS    PUFFL            ; Speicherplatz fuer Puffer
```

Wir vereinbaren, daß Bit 0 der Flags gelöscht wird, falls der Puffer voll ist, Bit 1 dagegen, wenn er leer ist. Dies führt zu folgenden Unterprogrammen zum Testen auf vollen beziehungsweise leeren Puffer:

```
VOLL:     PUSH    HL          ; Registerinhalt sichern
          LD      HL,FLAGS    ; Zeiger auf Flags generieren
          BIT     0,(HL)      ; Voll-Flag testen
          POP     HL          ; Register restaurieren
          RET

LEER:     PUSH    HL          ; Registerinhalt sichern
          LD      HL,FLAGS    ; Zeiger auf Flags generieren
          BIT     1,(HL)      ; Leer-Flag testen
          POP     HL          ; Register restaurieren
          RET
```

Außer den Puffer-Zeigern müssen wir natürlich jetzt auch die Flags initialisieren; da diese Operationen auch beim Füllen oder Leeren benötigt werden, schreiben wir dafür vier kurze Unterprogramme (VSETZ signalisiert vollen Puffer, VLOES nicht-vollen Puffer, LSETZ leeren Puffer, LLOES nicht-leeren Puffer):

```
VSETZ:    PUSH    HL          ; Registerinhalt sichern
          LD      HL,FLAGS    ; Zeiger auf Flags generieren
```

```
        RES     0,(HL)      ; Puffer als voll kennzeichnen
        POP     HL          ; Register restaurieren
        RET

VLOES:  PUSH    HL          ; Registerinhalt sichern
        LD      HL,FLAGS    ; Zeiger auf Flags generieren
        SET     0,(HL)      ; Puffer als nicht-voll
                            ; kennzeichnen
        POP     HL          ; Register restaurieren
        RET

LSETZ:  PUSH    HL          ; Registerinhalt sichern
        LD      HL,FLAGS    ; Zeiger auf Flags generieren
        RES     1,(HL)      ; Puffer als leer kennzeichnen
        POP     HL          ; Register restaurieren
        RET

LLOES:  PUSH    HL          ; Registerinhalt sichern
        LD      HL,FLAGS    ; Zeiger auf Flags generieren
        SET     1,(HL)      ; Puffer als nicht-leer
                            ; kennzeichnen
        POP     HL          ; Register restaurieren
        RET
```

Die Initialisierungssequenz lautet damit:

```
INIT:   PUSH    HL          ; Registerinhalt sichern
        CALL    LSETZ       ; Puffer als leer kennzeichnen
        CALL    VLOES       ; Puffer als nicht-voll
                            ; kennzeichnen
        LD      HL,ANFANG   ; Anfangsadresse des Puffers
        LD      (PZEIG),HL  ; Produzenten-Zeiger
                            ; initialisieren
        LD      (KZEIG),HL  ; Konsumenten-Zeiger
                            ; initialisieren
        POP     HL          ; Register restaurieren
        RET
```

Nun brauchen wir noch Unterprogramme, die den Produzenten-Zeiger beziehungsweise Konsumenten-Zeiger beim Überschreiten der Puffergrenze auf den Anfang des Puffers dirigieren:

```
PENDE:  PUSH    HL          ; Registerinhalte
        PUSH    DE          ; sichern
```

```
        LD      HL,(PZEIG)      ; Produzenten-Zeiger holen
        LD      DE,ENDE+1       ; Testwert laden
        OR      A               ; auf Ueberschreiten der
        SBC     HL,DE           ; Puffergrenze testen
        JP      NZ,PENDE1       ; Grenze nicht ueberschritten
        LD      HL,ANFANG       ; Produzenten-Zeiger auf Anfang
        LD      (PZEIG),HL      ; des Puffers dirigieren
PENDE1: POP     DE              ; Register
        POP     HL              ; restaurieren
        RET

KENDE:  PUSH    HL              ; Registerinhalte
        PUSH    DE              ; sichern
        LD      HL,(KZEIG)      ; Konsumenten-Zeiger holen
        LD      DE,ENDE+1       ; Testwert laden
        OR      A               ; auf Ueberschreiten der
        SBC     HL,DE           ; Puffergrenze testen
        JP      NZ,KENDE1       ; Grenze nicht ueberschritten
        LD      HL,ANFANG       ; Konsumenten-Zeiger auf Anfang
        LD      (KZEIG),HL      ; des Puffers dirigieren
KENDE1: POP     DE              ; Register
        POP     HL              ; restaurieren
        RET
```

Außerdem fehlt uns noch ein Unterprogramm, das feststellt, ob die beiden Puffer-Zeiger denselben Wert besitzen:

```
GLEICH: PUSH    HL              ; Registerinhalte
        PUSH    DE              ; sichern
        LD      HL,(PZEIG)      ; Produzenten-Zeiger holen
        LD      DE,(KZEIG)      ; Konsumenten-Zeiger holen
        OR      A               ; beide Zeiger
        SBC     HL,DE           ; vergleichen
        POP     DE              ; Register
        POP     HL              ; restaurieren
        RET
```

Das Füllen und Leeren des Puffers wird nun von folgenden Unterprogrammen erledigt:

```
FUELLE: CALL    VOLL            ; auf vollen Puffer testen
        RET     Z               ; Puffer voll, Fehler
        PUSH    HL              ; Registerinhalt sichern
        LD      HL,(PZEIG)      ; Produzenten-Zeiger holen
```

```
        LD      (HL),A      ; Zeichen im Puffer ablegen
        INC     HL          ; auf naechstes Zeichen zeigen
        LD      (PZEIG),HL  ; Produzenten-Zeiger abspeichern
        POP     HL          ; Register restaurieren
        CALL    LLOES       ; Puffer als nicht-leer
                            ; kennzeichnen
        PUSH    AF          ; Registerinhalt sichern
        CALL    PENDE       ; eventuell Produzenten-Zeiger
                            ; auf Anfang des Puffers richten
        CALL    GLEICH      ; auf vollen Puffer testen
        CALL    Z,VSETZ     ; Puffer als voll kennzeichnen
        POP     AF          ; Register restaurieren
        RET

LEERE:  CALL    LEER        ; auf leeren Puffer testen
        RET     Z           ; Puffer leer, Fehler
        PUSH    HL          ; Registerinhalt sichern
        LD      HL,(KZEIG)  ; Konsumenten-Zeiger holen
        LD      A,(HL)      ; Zeichen aus dem Puffer nehmen
        INC     HL          ; auf naechstes Zeichen zeigen
        LD      (KZEIG),HL  ; Konsumenten-Zeiger abspeichern
        POP     HL          ; Register restaurieren
        CALL    VLOES       ; Puffer als nicht-voll
                            ; kennzeichnen
        PUSH    AF          ; Registerinhalt sichern
        CALL    KENDE       ; eventuell Konsumenten-Zeiger
                            ; auf Anfang des Puffers richten
        CALL    GLEICH      ; auf leeren Puffer testen
        CALL    Z,LSETZ     ; Puffer als leer kennzeichnen
        POP     AF          ; Register restaurieren
        RET
```

Übungen

1. Überlege, wie die Unterprogramme für einen Ringpuffer mit 256 Bytes optimiert werden können, wenn das niederwertige Byte der Adresse ANFANG den Wert 00H besitzt.

2. Eine andere Implementierung des Ringpuffers läßt ein Zeichen des Puffers ungenutzt und spart dadurch die Flags für vollen und leeren Puffer. Bei welchen Operationen bringt dies Vorteile, bei welchen Nachteile in der Geschwindigkeit?

21
Tabellen

Eine *Tabelle* (engl. table) ist ein rechteckförmiges Schema von Werten, wobei die Werte innerhalb einer Spalte vom selben Typ sind. Die Werte innerhalb einer Zeile stehen in einem logischen Zusammenhang; eine Zeile einer Tabelle wird manchmal auch *Datensatz* genannt.

Tabellen sind ungemein mächtige Datenstrukturen; relationale Datenbanken sind vollständig aus Tabellen aufgebaut, in denen jeder Datensatz einem Eintrag in der Datenbank entspricht (dies kann zum Beispiel die Beschreibung eines Objekts sein).

21.1 Implementierung von Tabellen

Die logische Strukturierung einer Tabelle in Datensätze legt nahe, eine Tabelle als Feld von Verbunden zu implementieren; jeder Verbund stellt damit einen Datensatz dar. Wir kombinieren die für Felder und Verbunde erlernten Techniken und gelangen so zu verschiedenen Formen von Tabellen.

Eine Tabelle fester Länge kann ohne Deskriptor aufgebaut werden. Wir betrachten ein Beispiel: Unsere Tabelle soll über Länge, Breite und Dicke von Holzplatten einer Möbelfabrikation Auskunft geben. Jedes Produkt hat eine Produktnummer, die wir durch eine 16-Bit-Zahl codieren. Länge, Breite und Dicke geben wir in Millimeter an, ebenfalls als ganzzahlige 16-Bit- bzw. 8-Bit-Größen.

Jeder Datensatz hat damit folgende Struktur:

- Artikelnummer	2 Bytes
- Länge	2 Bytes
- Breite	2 Bytes
- Dicke	1 Byte

Die Tabelle könnte folgendermaßen aussehen:

```
PLATTE:
        DEFW    329         ; Produktnummer
        DEFW    1800        ; Laenge
        DEFW    450         ; Breite
        DEFB    12          ; Dicke

        DEFW    2391        ; Produktnummer
        DEFW    1260        ; Laenge
        DEFW    235         ; Breite
        DEFB    14          ; Dicke

        :
        :
        :

        DEFW    4138        ; Produktnummer
        DEFW    435         ; Laenge
        DEFW    240         ; Breite
        DEFB    8           ; Dicke
```

Die Länge der Tabelle wird irgendwo in den Zugriffsalgorithmen versteckt.

Eine weitere Möglichkeit, ohne Tabellendeskriptor auszukommen, besteht darin, hinter den letzten Eintrag der Tabelle einen Kennwert zu setzen, der sich von allen an dieser Stelle möglichen zulässigen Werten für Datensätze unterscheidet; der Kennwert kann den Platz eines ganzen Verbunds einnehmen, er kann aber auch kürzer sein, zum Beispiel ein Byte oder sogar ein Bit. Wir betrachten als Beispiel eine Tabelle von Namen, jeweils bestehend aus Vorname und Familiennname, die durch ein Null-Byte abgeschlossen ist:

```
NAMEN:
        DEFM    'Hans    '
        DEFM    'Mueller '
        DEFM    'Klaus   '
        DEFM    'Schulze '
        DEFM    'Heinrich'
        DEFM    'Seidl   '
        DEFB    0           ; Ende-Markierung der Tabelle
```

Als letzte Möglichkeit steht schließlich die Verwendung eines Deskriptors zur Wahl; dieser kann zum Beispiel unmittelbar vor dem ersten Tabelleneintrag stehen. Folgende Tabelle gibt eine Folge von Meßpunkten durch Paare von Meßwerten an; der Deskriptor besteht aus der Anzahl der Tabelleneinträge:

```
PUNKTE:   DEFB    5          ; Anzahl der Tabellenelemente

          DEFW    14756
          DEFW    19334

          DEFW    12390
          DEFW    9534

          DEFW    34679
          DEFW    22879

          DEFW    17998
          DEFW    6879

          DEFW    56782
          DEFW    31255
```

Es kann durchaus vorkommen, daß die Datensätze nur aus je einem Wert bestehen, zum Beispiel wenn wir Mengen durch Tabellen darstellen wollen. In diesen Fällen entartet die Tabelle zum gewöhnlichen Feld.

Übungen

1. Stelle eine Menge von echt positiven ganzzahligen Raumkoordinaten durch eine Tabelle dar. Benutze folgende Koordinaten:

12	44	21
16	39	55
22	117	98
3	39	44
16	57	83
32	61	9

2. Stelle folgende Buchstabenmenge als Tabelle dar: G, J, E, f, n, R, t, Z

3. Bringe folgende Liste von Rabatten in Tabellenform:

Abnahmemenge	Rabatt
50	2 %
100	3 %
200	5 %
500	8 %
1000	10 %

21.2 Indizierter Zugriff auf Tabellen

Die Feldstruktur einer Tabelle kann man explizit ausnutzen, um mittels eines Index auf einen bestimmten Tabelleneintrag zuzugreifen. Man darf die Elemente der Tabelle dann natürlich nicht beliebig anordnen, weil sonst der Zusammenhang zwischen Index und Tabellenelement verlorengeht; außerdem muß der Index des gesuchten Tabellenelements bekannt sein.

Wir betrachten als Beispiel den Zugriff auf einen bestimmten Punkt der Meßreihe aus dem vorhergehenden Unterkapitel. Der Index soll im A-Register stehen und ab Null gezählt werden; die erste Koordinate des Meßpunkts soll ins DE-Register, die zweite ins BC-Register gebracht werden. Das HL-Register zeigt auf die Tabelle. Die Längeninformation im Deskriptor wollen wir ausnutzen, um den Index auf seine Gültigkeit zu überprüfen:

```
        CP      (HL)        ; Gueltigkeit des Index pruefen
        JP      NC,FEHLER   ; ungueltiger Index
        INC     HL          ; auf ersten Tabelleneintrag zeigen
        EX      DE,HL       ; Basisadresse sichern
        LD      H,0         ; Index zu
        LD      L,A         ; Wort machen
        ADD     HL,HL       ; Relativadresse
        ADD     HL,HL       ; berechnen
        ADD     HL,DE       ; Adresse des gesuchten Eintrags
        LD      E,(HL)      ; erste
        INC     HL          ; Koordinate
        LD      D,(HL)      ; holen
        INC     HL          ; auf zweite Koordinate zeigen
        LD      C,(HL)      ; zweite
        INC     HL          ; Koordinate
        LD      B,(HL)      ; holen
```

Auch einen Stapel kann man als Tabelle auffassen. Das oberste Element ist dabei zweckmäßigerweise der letzte Tabelleneintrag; so wächst die Tabelle zu größeren Adressen hin. Im Deskriptor wird zweckmäßigerweise der Stapel-Zeiger vermerkt. Wenn unser Stapel Elemente vom Typ »Byte« aufnimmt und der Stapel-Zeiger der Struktur unter der Adresse STAPEL abgespeichert ist (dies braucht nicht unmittelbar vor dem ersten Tabelleneintrag sein), so realisieren folgende Unterprogramme die Operationen PUSH und POP (das A-Register dient zur Aufnahme eines Stapel-Elements):

```
PUSH:   LD      HL,(STAPEL)     ; Stapel-Zeiger holen
        INC     HL              ; auf freien Speicherplatz zeigen
        LD      (HL),A          ; Element ablegen
        LD      (STAPEL),HL     ; Stapel-Zeiger abspeichern
        RET
```

```
POP:    LD      HL,(STAPEL)   ; Stapel-Zeiger holen
        LD      A,(HL)        ; Element entnehmen
        DEC     HL            ; auf oberstes Element
                              ; des Stapels zeigen
        LD      (STAPEL),HL   ; Stapel-Zeiger abspeichern
        RET
```

Auf Fehlerbehandlungen (Stapelüberlauf, Stapelunterlauf) und Retten des HL-Registers haben wir dabei verzichtet.

Wem das lieber ist, der kann den Stapel auch als hängenden Stapel realisieren oder die Konvention für den Stapel-Zeiger ändern.

In ähnlicher Weise läßt sich auch ein Puffer, dessen Elemente nicht vom Typ »Byte« sind, durch eine Tabelle realisieren. Die neu hinzukommenden Elemente werden vom Produzenten ans Ende der Tabelle angehängt. Der Konsument entnimmt Elemente vom Anfang der Tabelle; dadurch freiwerdender Speicherplatz muß irgendwann durch Verschieben der restlichen Tabelleneinträge wieder nutzbar gemacht werden (dies kann zum Beispiel jedesmal nach dem Entfernen eines Elements geschehen).

Übungen

1. Bei einem Quiz erzielten die Kandidaten folgende Punktwerte:

Huber	12
Meier	28
Schulz	21
Gruber	18
Jung	14
Weiss	21

   Diese Informationen sollen als indizierte Tabelle abgespeichert werden. Schreibe dann ein Unterprogramm, das zu vorgegebenem Index den Anfangsbuchstaben des Namens und die erreichte Punktzahl liefert.

2. Realisiere einen Puffer von Worten durch eine Tabelle (Datenstruktur und Zugriffsmechanismen).

21.3 Schlüssel-orientierter Zugriff auf Tabellen

Beim schlüssel-orientierten Zugriff auf eine Tabelle spielt die Reihenfolge der Elemente keine Rolle. Ein bestimmtes Element wird dadurch ausgewählt, daß für eine oder mehrere Komponenten des Verbunds Werte vorgegeben werden. In der ersten Tabelle aus Unterkapitel 21.1

könnte zum Beispiel eine bestimmte Produktnummer vorgegeben sein, oder eine bestimmte Kombination von Länge, Breite und Dicke.

Die Suchoperation kann fehlschlagen, wenn kein Element vorhanden ist, das die Vorgaben erfüllt; andererseits kann es auch vorkommen, daß die Beschreibung auf mehrere Elemente der Tabelle zutrifft.

Allgemeiner kann man statt Werten auch Relationen zwischen bestimmten Komponenten vorgeben; und statt eines eindeutig bestimmten Eintrags kann man auch die Menge aller Einträge bestimmen, die der Beschreibung entsprechen. In einem Katalog für Polstermöbel könnten zum Beispiel folgende Daten aufgelistet sein:

Artikelname
Artikelnummer
Artikelbezeichnung
Farbe
Material
Preis

Eine mögliche Suchoperation wäre dann: Liefere alle Artikelnummern von Einträgen mit der Bezeichnung »Couch«, der Farbe Schwarzbraun oder Rostbraun, aus Rohleder, mit einem Preis nicht über 3400,– DM.

Die Vorgehensweise bei einer solchen Suchoperation ist folgende: Man stellt sich einen Zeiger auf das erste Element der Tabelle bereit und prüft, ob dieses die geforderten Eigenschaften hat. Wenn dies der Fall ist, so wird das Element (oder die benötigten Komponenten) aus der Tabelle kopiert. Anschließend wird in beiden Fällen (durch Addition der festen Länge eines Tabellenelements) der Zeiger auf das nächste Element der Tabelle fortgeschaltet, bis das Ende der Tabelle erreicht ist. Die Feldstruktur der Tabelle wird dabei nur zum Fortschalten der Basis-Adresse des Verbunds benutzt.

Hierzu ein Beispiel: Gegeben sei eine Tabelle, in der für eine Reihe von Personen drei Kenngrößen festgehalten werden:

Alter (in Jahren)	1 Byte
Gewicht (in kg)	1 Byte
Größe (in cm)	1 Byte

Als Ende-Markierung für die Tabelle wählen wir ein Null-Byte, weil das Alter 0 nicht vorkommen kann. Die Tabelle sieht zum Beispiel folgendermaßen aus:

```
PERSON:
        DEFB    26          ; Alter
        DEFB    93          ; Gewicht
        DEFB    188         ; Groesse
```

```
        DEFB    39          ; Alter
        DEFB    65          ; Gewicht
        DEFB    176         ; Groesse
        :
        :
        :
        DEFB    19          ; Alter
        DEFB    80          ; Gewicht
        DEFB    182         ; Groesse
        DEFB    0           ; Ende-Markierung
```

Nun bauen wir eine zweite Tabelle PERS2 mit gleicher Struktur auf, welche die Kenngrößen derjenigen Personen enthält, die zwischen 18 und 35 Jahren (einschließlich) alt und kleiner als 175 cm sind:

```
        LD      IX,PERSON   ; Zeiger auf erste Tabelle
        LD      IY,PERS2    ; Zeiger auf zweite Tabelle
TEST:   LD      A,(IX+0)    ; Alter holen
        OR      A           ; auf Null-Byte testen
        JP      Z,FERTIG    ; zweite Tabelle komplett
        CP      18          ; Alter testen
        JP      C,WEITER    ; juenger als 18
        CP      36          ; Alter testen
        JP      NC,WEITER   ; aelter als 35
        LD      A,(IX+2)    ; Groesse holen
        CP      175         ; Groesse testen
        JP      NC,WEITER   ; nicht kleiner als 175
        LD      A,(IX+0)    ; Alter holen
        LD      (IY+0),A    ; Alter kopieren
        LD      A,(IX+1)    ; Gewicht holen
        LD      (IY+1),A    ; Gewicht kopieren
        LD      A,(IX+2)    ; Groesse holen
        LD      (IY+2),A    ; Groesse kopieren
WEITER: INC     IX          ; auf naechstes
        INC     IX          ; Element der ersten
        INC     IX          ; Tabelle zeigen
        INC     IY          ; auf naechstes
        INC     IY          ; Element der zweiten
        INC     IY          ; Tabelle zeigen
        JP      TEST        ; Rest der Tabelle bearbeiten
FERTIG: LD      (IY+0),0    ; Ende der zweiten Tabelle
                            ; markieren
```

Das Programm ist nicht optimiert! Versuche ein optimiertes Programm zu schreiben!

Ein spezieller Fall liegt vor, wenn jeder Tabelleneintrag nur aus einer Komponente besteht; wir haben dann eine Folge von Elementen eines bestimmten Typs vorliegen, die wir zum Beispiel als Menge interpretieren können. Auch hierzu ein Beispiel: Wir betrachten Tabellen, die Teilmengen der Menge von ASCII-Zeichen darstellen; die Teilmengen sollen ungeordnet sein. Die Kardinalität der Teilmengen ist dem Deskriptor zu entnehmen. Zwei solche Teilmengen wären:

```
MENGE1:  DEFB    4          ; Kardinalitaet 4
         DEFB    'H'
         DEFB    '*'
         DEFB    'a'
         DEFB    'Z'
MENGE2:  DEFB    3          ; Kardinalitaet 3
         DEFB    '+'
         DEFB    '-'
         DEFB    ' '
```

Wir zeigen exemplarisch einige Mengenoperationen auf dieser Struktur:

Mit Hilfe des Unterprogramms ELEM stellen wir fest, ob ein im A-Register stehendes Zeichen in der Menge enthalten ist, auf die das HL-Register zeigt. Wenn ja, so soll das Null-Flag gesetzt werden:

```
ELEM:    PUSH    BC         ; Registerinhalt sichern
         LD      C,(HL)     ; Tabellenlaenge beschaffen
         INC     HL         ; auf erstes Element
                            ; der Menge zeigen
         INC     C          ; Laenge auf
         DEC     C          ; Null testen
         JP      Z,LEER     ; leere Menge
         LD      B,0        ; Laenge zu Wort machen
         CPIR               ; Zeichen in Menge suchen
         POP     BC         ; Register restaurieren
         RET
LEER:    INC     C          ; Null-Flag loeschen
         POP     BC         ; Registerinhalt restaurieren
         RET
```

Das Hinzufügen eines Elements, das im A-Register steht, zu einer Menge, auf die das HL-Register zeigt, wird durch folgendes Unterprogramm realisiert:

```
HINEIN:  PUSH    HL         ; Registerinhalt sichern
         CALL    ELEM       ; Feststellen, ob Zeichen
                            ; bereits in der Menge
                            ; enthalten ist
```

```
        JP      Z,FERTIG    ; Zeichen schon in der Menge
        LD      (HL),A      ; HL zeigt direkt hinter Menge,
                            ; Zeichen hinzufuegen
        POP     HL          ; Register restaurieren
        INC     (HL)        ; Laengenangabe aktualisieren
        RET
FERTIG: POP     HL          ; Register restaurieren
        RET
```

Achte besonders auf den Seiteneffekt von ELEM: Falls das gesuchte Zeichen noch nicht in der Menge enthalten ist, zeigt das HL-Register nach Rückkehr aus ELEM auf das nächste Zeichen direkt hinter der Menge.

Als letztes betrachten wir die Vereinigung der beiden Mengen, auf die das HL-Register beziehungsweise DE-Register zeigt; die zweite Menge soll dabei in die erste Menge eingefügt werden:

```
VEREIN: PUSH    AF          ; Register-
        PUSH    BC          ; inhalte
        PUSH    DE          ; sichern
        LD      A,(DE)      ; Kardinalitaet der zweiten
        LD      B,A         ; Menge holen
        INC     B           ; Schleife abweisend machen
        JP      VER3        ; in Schleife einspringen
VER2:   INC     DE          ; auf naechstes Element
                            ; der zweiten Menge zeigen
        LD      A,(DE)      ; Element holen
        CALL    HINEIN      ; Element zur ersten Menge
                            ; hinzufuegen
VER3:   DJNZ    VER2        ; gesamte Menge durchgehen
        POP     DE          ; alle
        POP     BC          ; Register
        POP     AF          ; restaurieren
        RET
```

Übungen

1. Schreibe folgende Liste von Paaren (Vorname, Alter) als Tabelle:

Petra	25
Hans	28
Otto	27
Hanna	29

Klaus	22
Inge	27
Heinz	27
Claudia	26
Peter	22

Entwickle ein Unterprogramm, das zu vorgegebenem Alter die Tabelle derjenigen Personen erstellt, die jünger sind.

2. Es soll eine Tabelle mit folgender Struktur der Einträge erstellt werden:

Bit 0	0 = männlich	1 = weiblich
Bit 1	0 = verheiratet	1 = ledig
Bit 2/3	00 = Arbeiter	01 = Angestellter
	10 = Beamter	11 = Selbständig

Schreibe ein Programm, das aus einer solchen Tabelle ermittelt, wie viele Personen männlich beziehungsweise weiblich sind, wie viele verheiratet usw.

3. Realisiere für die zuletzt gegebene Darstellung von Mengen die Operationen »Entfernen eines Elements« und »Differenz zweier Mengen«.

22
Alternativen

Eine *Alternative* realisiert die Auswahl eines Programmstücks aus einer Menge von alternativen (daher die Bezeichnung) Programmstücken. Dabei wird genau eines der vorgegebenen Programmstücke der Alternative ausgeführt. Die Auswahl kann von verschiedenen Kriterien bestimmt werden, die wir in den folgenden drei Unterkapiteln kennenlernen werden.

Alternativen sind in gewisser Hinsicht eine spezielle Form von Verzweigungen; sie können in der Tat stets durch Verzweigungsketten modelliert werden. Direkte Implementationen als Alternativen sind aber meist effizienter als ihre ersatzweise Darstellung durch Verzweigungen.

Höhere Programmiersprachen verfügen meist über ein Alternativen-Konstrukt; in PASCAL zum Beispiel ist dies das CASE-Konstrukt.

22.1 Berechnete Sprünge

Ein berechneter Sprung ist eine Alternative, bei der die Auswahl des Programmstücks von einem Index abhängt. Wir wollen hier nur Indizes betrachten, die ab Null fortlaufend gezählt werden (Verallgemeinerungen sind leicht möglich). Die Programmstücke sind von 0 bis zu einer Zahl n durchnumeriert; ausgeführt werden soll das dem Index zugeordnete Programmstück.

Die Alternative besteht nun aus zwei Teilen: Wir speichern die Anfangsadressen der Programmstücke als Feld von Worten so ab, daß das i-te Feldelement die Adresse des i-ten Programmstücks angibt; außerdem benötigen wir eine Routine für die Auswahl der Adresse und die Aktivierung des dadurch gegebenen Programmstücks.

Wir nehmen zunächst an, daß der Index stets gültig ist, und daß der Zugriffsmechanismus der Alternative als Programmstück implementiert werden soll, durch welches das gewünschte Programmstück angesprungen wird; letzteres sorgt dann selbst für eine geeignete Fortsetzung des Programms.

Das Feld von Sprungadressen, das wir getrost als Tabelle interpretieren können und das deshalb auch meist *Sprungtabelle* genannt wird, könnte etwa folgendermaßen aussehen:

```
SPRUNG:  DEFW    ADR0      ; Sprungadresse zum Index 0
         DEFW    ADR1      ; Sprungadresse zum Index 1
         DEFW    ADR2      ; Sprungadresse zum Index 2
         :
         :
         :
         DEFW    ADRN      ; Sprungadresse zum Index n
```

Das Programmstück für das Berechnen der Sprungadresse aus dem Index kennen wir im Prinzip schon (den Index erwarten wir im A-Register):

```
         LD      HL,SPRUNG ; Basisadresse der Sprungtabelle
                           ; laden
         LD      D,0       ; Index zu
         LD      E,A       ; Wort erweitern
         ADD     HL,DE     ; absolute Adresse der
         ADD     HL,DE     ; Sprungadresse berechnen
         LD      E,(HL)    ; LSB der Sprungadresse holen
         INC     HL        ; auf MSB der Sprungadresse
                           ; zeigen
         LD      D,(HL)    ; MSB der Sprungadresse holen
         EX      DE,HL     ; Sprungadresse verfuegbar machen
         JP      (HL)      ; gewuenschtes Programmstueck
                           ; anspringen
```

Eine andere Form des berechneten Sprungs ist die sogenannte *Sprungleiste*. Dabei wird nicht eine Sprungtabelle angelegt, sondern eine Folge von Sprungbefehlen, die zu den gewünschten Programmstücken führen. Der Objekt-Code dieser Sprungbefehle bildet eine lückenlose Folge im Speicher, so daß die Adresse des richtigen Sprungbefehls berechnet werden kann, wenn man über die Anfangsadresse der Sprungleiste und über den Index verfügt. Wenn absolute Sprünge benutzt werden, ist die Relativadresse des Sprungbefehls das Dreifache des Index, bei relativen Sprüngen das Doppelte. Wir zeigen ein Beispiel mit absoluten Sprüngen, zunächst die eigentliche Sprungleiste:

```
LEISTE:  JP      ADR0      ; Programmstueck zum Index 0
                           ; anspringen
         JP      ADR1      ; Programmstueck zum Index 1
                           ; anspringen
```

```
        :
        :
        :
        JP          ADRN        ; Programmstueck zum Index n
                                ; anspringen
```

Nun die Berechnung der Adresse des entsprechenden Sprungbefehls; der Algorithmus ist dem vorhergehenden sehr ähnlich. Wir nehmen diesmal an, daß das Programmstück als Unterprogramm aufgerufen werden soll und daß alle Programmstücke der Alternative als abgeschlossene Unterprogramme ausgeführt sind (zu den dabei verwendeten Techniken vergleiche das Kapitel »Unterprogramme«):

```
ALTERN: LD          HL,LEISTE   ; Basisadresse der Sprungleiste
                                ; laden
        LD          D,0         ; Index zu
        LD          E,A         ; Wort erweitern
        ADD         HL,DE       ; absolute Adresse des
        ADD         HL,DE       ; Sprungbefehls
        ADD         HL,DE       ; berechnen
        JP          (HL)        ; gewuenschten Sprungbefehl
                                ; anspringen
```

Ein möglicher Aufruf wäre:

```
        :
        :
        :
        LD          A,2         ; Index mit 2 besetzen
        CALL        ALTERN      ; Unterprogramm mit der
                                ; Funktionsnummer 2 ausfuehren
        :
        :
        :
```

22.2 Wert-gesteuerte Alternativen

Bei den berechneten Sprüngen haben wir Gebrauch von einem Index gemacht, dem ein Programmstück eindeutig zugeordnet war. Diesen Index muß man normalerweise aus anderen Daten erst berechnen; meist wird er einer Tabelle durch schlüssel-orientierten Zugriff entnommen. Die Berechnung des Index und die anschließende Auswahl des Programmstücks kann man zu einer wert-gesteuerten Alternative zusammenfassen.

Die Auswahl eines Programmstücks in einer wert-gesteuerten Alternative geschieht mittels eines Such-Werts, der mit vorgegebenen Werten in einer Tabelle verglichen wird. Zu jedem Wert enthält die Tabelle eine Sprungadresse. Stimmt der Such-Wert mit einem der Werte der Tabelle überein, so wird die zugehörige Adresse angesprungen. Auch hier können wir wieder vorsehen, daß bei Fehlschlagen des Suchens ein Ausnahme-Programmstück ausgeführt wird.

Wir gestalten die Tabelle nun so, daß zu Beginn die Anzahl der Einträge steht, gefolgt von den einzelnen Einträgen, die aus jeweils einem Vergleichs-Wert und einer Sprungadresse bestehen; eventuell wird an diese eigentliche Tabelle noch die Adresse einer Ausnahme-Routine angehängt. Als Beispiel nehmen wir Werte vom Typ »Byte«:

```
SPRUNG:  DEFB    4          ; Tabelle mit 4 Eintraegen
         DEFB    '+'        ; erster Vergleichs-Wert
         DEFW    PLUS       ; zugehoerige Sprungadresse
         DEFB    '-'        ; zweiter Vergleichs-Wert
         DEFW    MINUS      ; zugehoerige Sprungadresse
         DEFB    '*'        ; dritter Vergleichs-Wert
         DEFW    MULT       ; zugehoerige Sprungadresse
         DEFB    '/'        ; vierter Vergleichs-Wert
         DEFW    DIV        ; zugehoerige Sprungadresse
         DEFW    FEHLER     ; Sprungadresse der
                            ; Ausnahme-Routine
```

Wie wir bereits im vorhergehenden Unterkapitel sehen konnten, unterscheiden sich verschiedene Anwendungen eines Typs von Alternativen nur durch die verwendeten Tabellen. Wir schreiben unsere Ansprung-Routine deshalb als Unterprogramm, dem bei der Aktivierung der Such-Wert (im A-Register) und ein Zeiger (im HL-Register) auf die richtige Tabelle übergeben wird; alle Programmstücke der Alternative sollen ebenfalls als Unterprogramme ausgeführt sein:

```
ALTERN:  PUSH    BC         ; Registerinhalt sichern
         LD      B,(HL)     ; Anzahl der Eintraege laden
         INC     HL         ; auf ersten Vergleichs-Wert
                            ; zeigen
         INC     B          ; Schleife abweisend
         JP      TEST       ; machen
SUCHE:   CP      (HL)       ; Such-Wert vergleichen
         INC     HL         ; auf zugehoerige Adresse zeigen
         JP      Z,GEFUND   ; Such-Wert gefunden,
                            ; zugehoeriges Unterprogramm
                            ; anspringen
         INC     HL         ; auf naechsten Vergleichs-Wert
         INC     HL         ; zeigen
TEST:    DJNZ    SUCHE      ; ganze Tabelle durchsuchen
```

```
GEFUND:  LD      C,(HL)      ; LSB der Sprungadresse holen
         INC     HL          ; auf MSB der Sprungadresse
                             ; zeigen
         LD      H,(HL)      ; MSB der Sprungadresse holen,
                             ; Zeiger auf Tabelle wird
                             ; nicht mehr benoetigt
         LD      L,C         ; Sprungadresse ins HL-Register
                             ; bringen
         POP     BC          ; Register restaurieren
         JP      (HL)        ; Unterprogramm anspringen
```

Ein Aufruf des Unterprogramms, der die oben aufgeführte Tabelle benutzt, wäre dann:

```
         :
         :
         :
         LD      A,'*'       ; Vergleichswert laden
         LD      HL,SPRUNG   ; Adresse der Tabelle laden
         CALL    ALTERN      ; gewuenschtes Unterprogramm
                             ; ausfuehren
         :
         :
         :
```

Übungen

1. Im B-Register und C-Register stehe je eine vorzeichenlose ganze Zahl. Im A-Register stehe eines der Relationszeichen <, =, >. Schreibe eine Alternative, die feststellt, ob die Relation <B> <A> <C> wahr (Übertrag-Flag setzen) oder falsch (Übertrag-Flag löschen) ist; gemeint ist die Relation, die entsteht, wenn vor das Relationszeichen, das sich im A-Register befindet, der Wert aus dem B-Register geschrieben wird, dahinter der Wert aus dem C-Register.

2. Erweitere die Routine aus Aufgabe 1 so, daß das Null-Flag genau dann gesetzt ist, wenn im A-Register wirklich eines der drei Relationszeichen steht, sonst aber gelöscht.

22.3 Attribut-gesteuerte Alternativen

Bei den wert-gesteuerten Alternativen haben wir auf der Suche nach einem mit dem Such-Wert übereinstimmenden Vergleichs-Wert eine Reihe von Vergleichen durchlaufen. Dieses Verfahren läßt sich zu Attribut-gesteuerten Alternativen verallgemeinern, bei denen der Reihe

nach eine Folge von Attributen - Relationen zwischen bestimmten Registern und Speicherzellen - geprüft wird. Jedem Attribut ist ein Programmstück zugeordnet, das ausgeführt wird, falls das Attribut zutrifft. Es soll dabei genau ein Programmstück ausgeführt werden, nämlich das zum ersten zutreffenden Attribut gehörende, falls es ein solches gibt, ansonsten ein Ausnahme-Programmstück.

Wir sehen uns ein Beispiel für den Einsatz einer Attribut-gesteuerten Alternative an: In Übung 3 von Kapitel 9.4 wurde eine Cursor-Steuerung beschrieben; dabei konnte es vorkommen, daß die gewünschte Bewegung nicht durchgeführt werden kann, weil der Cursor sonst den zulässigen Bereich des Bildschirms verlassen würde. Wir vereinfachen das Beispiel dahingehend, daß wir nur die Bewegungsrichtung aufwärts/abwärts betrachten. Die Alternative würde dann aus folgenden Attributen und zugeordneten Aktionen bestehen (wenn eine Bewegung nicht durchführbar ist, soll ein Piepsen ertönen):

- Bewegung aufwärts, Cursor am oberen Rand:
  Piepsen
- Bewegung aufwärts, Cursor nicht am oberen Rand:
  Cursor um eine Zeile aufwärts bewegen
- Bewegung abwärts, Cursor am unteren Rand:
  Piepsen
- Bewegung abwärts, Cursor nicht am unteren Rand:
  Cursor um eine Zeile abwärts bewegen
- Code unzulässig:
  Piepsen

Wir fassen die Adressen der Programmstücke für die Prüfung der Attribute und ihrer zugeordneten Programmstücke in einer Tabelle zusammen. Zu Beginn steht die Anzahl der Attribute. Für jedes Attribut folgt dann die Adresse des Programmstücks zur Prüfung des Attributs und die Adresse des zugeordneten Programmstücks. Als letztes kommt noch die Adresse des Ausnahme-Programmstücks. Ein Beispiel für eine solche Tabelle wäre damit:

```
CURSOR:   DEFB    4          ; 4 Attribute in Alternative

          DEFW    AUFOB      ; Adresse der Routine, die
                             ; prueft, ob Bewegung aufwaerts
                             ; gewuenscht und Cursor am
                             ; oberen Rand
          DEFW    PIEPS      ; Adresse der Routine zum
                             ; Aktivieren des Pieps

          DEFW    AUFNOB     ; Adresse der Routine, die
                             ; prueft, ob Bewegung aufwaerts
                             ; gewuenscht und Cursor
                             ; nicht am oberen Rand
```

```
        DEFW    AUFZEI      ; Adresse der Routine, die
                            ; Cursor eine Zeile hochschiebt

        DEFW    ABUNT       ; Adresse der Routine, die
                            ; prueft, ob Bewegung abwaerts
                            ; gewuenscht und Cursor am
                            ; unteren Rand
        DEFW    PIEPS       ; Adresse der Routine zum
                            ; Aktivieren des Pieps

        DEFW    ABNUNT      ; Adresse der Routine, die
                            ; prueft, ob Bewegung abwaerts
                            ; gewuenscht und Cursor
                            ; nicht am unteren Rand
        DEFW    ABZEI       ; Adresse der Routine, die
                            ; Cursor eine Zeile tiefer setzt

        DEFW    PIEPS       ; Adresse der Routine zum
                            ; Aktivieren des Pieps
                            ; (Ausnahme-Routine)
```

Wir wollen annehmen, daß alle Programmstücke zur Prüfung der Attribute und alle zugeordneten Programmstücke als Unterprogramme ausgeführt sind. Jedes Unterprogramm zur Prüfung eines Attributs soll durch das Übertrag-Flag signalisieren, ob das Attribut zutrifft; dabei steht gesetztes Übertrag-Flag für ein zutreffendes Attribut. BC-Register und DE-Register dürfen ihren Wert nicht ändern. Die folgende Routine wird als Unterprogramm aufgerufen und realisiert diejenige Alternative, auf deren Tabelle das HL-Register zeigt:

```
ATTRIB: PUSH    BC          ; Registerinhalte
        PUSH    DE          ; sichern
        LD      B,(HL)      ; Anzahl der Attribute holen
        INC     HL          ; auf Adresse der Routine zur
                            ; Pruefung des ersten Attributs
                            ; zeigen
        INC     B           ; Schleife abweisend
        JP      TEST        ; machen
SUCHE:  LD      DE,WEITER   ; Adresse des Wiedereintritts-
                            ; punkts laden
        PUSH    DE          ; und auf den Stapel bringen
        LD      E,(HL)      ; Adresse der Routine
        INC     HL          ; zur Pruefung des
        LD      D,(HL)      ; Attributs holen
        INC     HL          ; auf Adresse des zugeordneten
```

```
                                        ; Programmstuecks zeigen
            EX          DE,HL           ; Zeiger sichern,
                                        ; Sprungadresse verfuegbar machen
            JP          (HL)            ; Routine zur Prüfung des
                                        ; Attributs aufrufen

; Wiedereintrittspunkt

WEITER:     EX          DE,HL           ; Zeiger restaurieren
            JP          C,GEFUND        ; Attribut trifft zu,
                                        ; entsprechende Routine
                                        ; ausfuehren
            INC         HL              ; Adresse der Routine
            INC         HL              ; ueberlesen
TEST:       DJNZ        SUCHE           ; alle Alternativen bearbeiten
GEFUND:     LD          C,(HL)          ; Adresse
            INC         HL              ; der zugeordneten
            LD          H,(HL)          ; Routine
            LD          L,C             ; holen
            POP         DE              ; Register
            POP         BC              ; restaurieren
            JP          (HL)            ; Routine anspringen
```

Übungen

1. Führe das in diesem Unterkapitel gebrachte Beispiel vollständig aus.

23
Verzeigerte Strukturen

Verzeigerte Datenstrukturen bestehen aus einer Menge von Elementen, die als Verbunde ausgeführt sind; jeder Verbund enthält ein oder mehrere Komponenten, die den Wert des Elements darstellen, und ein oder mehrere Zeiger auf andere Elemente der Datenstruktur.

Die wichtigsten verzeigerten Strukturen sind Listen, Bäume und Graphen.

23.1 Listen

Eine Liste - genauer *lineare Liste* - ist eine verzeigerte Datenstruktur, bei der die Verzeigerung so gestaltet ist, daß jedes Element (bis auf eventuelle Anfangs- und Endelemente der Liste) genau einen Vorgänger und einen Nachfolger besitzt; die Elemente der Liste können also in eine lineare Reihenfolge gebracht werden. Numerieren wir die Elemente der Liste von 1 bis n durch, so besitzt das i-te Element höchstens Zeiger auf das (i−1)-te und (i+1)-te Element (modulo n gerechnet).

In einer *einfach verketteten linearen Liste* besitzt jedes Element genau einen Zeiger auf ein anderes Element; der Zeiger des i-ten Elements ist für $i < n$ auf das (i+1)-te Element gerichtet. Der Zeiger des n-ten Elements zeigt ins »Leere«; dies bedeutet, daß er keinen Wert besitzt, welcher der Adresse eines anderen Elements der Liste entsprechen könnte. Dieser - quasi ungültige - Wert eines Zeigers wird meist durch NIL (lat. nil = nichts) bezeichnet. Oft wird NIL als 0000H vereinbart; dies geht immer dann, wenn der Datenspeicher erst ab einer höheren Adresse als 0000H beginnt (beginnt der Datenspeicher ab der Adresse 0000H, so eignet sich meist für NIL der Wert FFFFH). Wir werden alle folgenden Routinen so schreiben, daß NIL als Konstante eingeht, also im Prinzip jeden beliebigen Wert haben kann.

Natürlich brauchen wir auch noch einen Einstieg in die Liste, also einen Zeiger auf das erste Element. Eine einfach verkettete lineare Liste stellen wir uns also folgendermaßen vor:

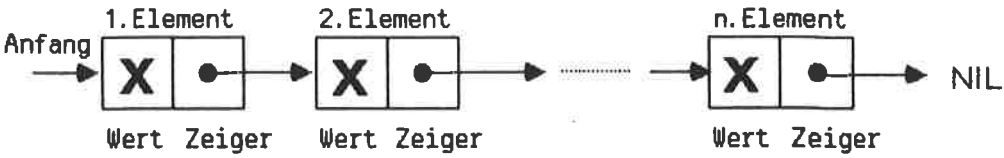


Bild 23.1. *Darstellung einer einfach verketteten linearen Liste*

Wenn der Zeiger, der auf die Liste zeigt, den Wert NIL besitzt, so ist die Liste leer, enthält also keine Elemente.

Wir betrachten nun als Beispiel eine lineare Liste, deren Elemente jeweils einen Wert von Typ »Byte« tragen. Die Elemente sind damit Verbunde, bestehend aus einem Byte (dem Wert) und einem Wort (dem Zeiger). Die Elemente können beliebig über den gesamten Datenspeicher verstreut sein, zum Beispiel:

```
ANFANG:  DEFW    ELEM1    ; Zeiger auf erstes Element
ELEM2:   DEFB    'B'      ; Wert des zweiten Elements
         DEFW    ELEM3    ; Zeiger auf drittes Element
ELEM3:   DEFB    'C'      ; Wert des dritten Elements
         DEFW    NIL      ; Zeiger ins Leere
ELEM1:   DEFB    'A'      ; Wert des ersten Elements
         DEFW    ELEM2    ; Zeiger auf zweites Element
```

Diese Liste besteht aus drei Elementen.

Es gibt nun diverse Operationen auf Listen, die in jeder Anwendung vorkommen. Wir wollen im Folgenden stets annehmen, daß eine Liste durch den Zeiger auf ihr erstes Element gegeben ist; diesen Zeiger wollen wir stets im HL-Register erwarten.

Eine Grundoperation ist das Testen eines Zeigers auf NIL (in diesem Fall soll das Null-Flag gesetzt werden):

```
ISTNIL:  EX      DE,HL    ; Zeiger sichern
         PUSH    HL       ; Registerinhalt sichern
         LD      HL,NIL   ; NIL laden
         OR      A        ; Uebertrag-Flag loeschen
         SBC     HL,DE    ; Vergleich durchfuehren
         POP     HL       ; Register restaurieren
         EX      DE,HL    ; Zeiger restaurieren
         RET
```

Den Wert eines Elements können wir bearbeiten, indem wir den Zeiger auf das Element – der ja auch ein Zeiger auf den Wert ist – benutzen; wir werden darauf näher in den folgenden drei Unterkapiteln eingehen.

Eine wichtige Operation besteht darin, aus dem Zeiger auf ein Element den Zeiger auf den Nachfolger des Elements zu generieren. Dabei kann es vorkommen, daß der gegebene Zeiger den Wert NIL hat, also auf gar kein Element zeigt. In diesem Fall existiert kein Nachfolger, was wir durch Setzen des Null-Flags kennzeichnen. Der neue Zeiger soll im DE-Register zurückgeliefert werden:

```
NACHFO:   CALL    ISTNIL    ; Zeiger auf NIL testen
          RET     Z         ; ungueltiger Zeiger
          INC     HL        ; auf Adresse des Nachfolgers
                            ; zeigen
          LD      E,(HL)    ; Adresse des
          INC     HL        ; Nachfolgers
          LD      D,(HL)    ; holen
          DEC     HL        ; Zeiger
          DEC     HL        ; restaurieren
          RET
```

Weitere Operationen auf Listen werden wir in den folgenden Unterkapiteln kennenlernen.

Sind die Werte der Elemente einer Liste selbst komplexe Datenstrukturen, so speichert man in den Listenelementen nicht die Werte selbst, sondern Zeiger darauf ab. Wenn jedes Listenelement einen Wert bezeichnet, ergibt sich dabei folgendes Schema:

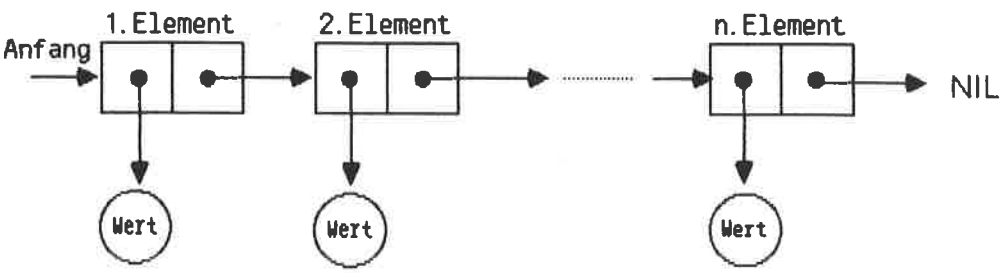


Bild 23.2. *Darstellung einer Liste von Zeigern*

Für bestimmte Anwendungen ist es sinnvoll, das letzte Element einer einfach verketteten linearen Liste wieder mit dem ersten Element der Liste zu verbinden. Es entsteht dadurch eine *zyklisch verkettete lineare Liste*:

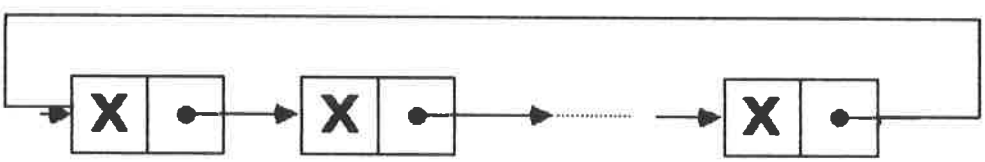


Bild 23.3. *Darstellung einer zyklisch verketteten linearen Liste*

Wir formen die oben aufgestellte einfach verkettete lineare Liste in eine zyklisch verkettete lineare Liste um:

```
ANFANG:  DEFW    ELEM1    ; Zeiger auf erstes Element
ELEM2:   DEFB    'B'      ; Wert des zweiten Elements
         DEFW    ELEM3    ; Zeiger auf drittes Element
ELEM3:   DEFB    'C'      ; Wert des dritten Elements
         DEFW    ELEM1    ; Zeiger auf erstes Element
ELEM1:   DEFB    'A'      ; Wert des ersten Elements
         DEFW    ELEM2    ; Zeiger auf zweites Element
```

Die Operationen auf einer zyklisch verketteten linearen Liste sind im Prinzip die gleichen wie auf einer einfach verketteten linearen Liste; allerdings braucht bei einem gültigen Zeiger auf ein Element nicht geprüft werden, ob ein Nachfolger existiert, da dies – außer in der leeren Liste – stets garantiert ist.

Wir geben noch ein anschauliches Beispiel für den Einsatz einer zyklisch verketteten linearen Liste: In einem Computersystem sind mehrere externe Geräte zu bedienen, die in regelmäßigen Abständen nacheinander an der Reihe sind. Die Kenndaten der Geräte stellen die Werte einer zyklisch verketteten linearen Liste dar. Soll das nächste Gerät bedient werden, so geht man einfach zum nächsten Element der Liste über. Da die Geräte im Prinzip gleichberechtigt sind, braucht es keinen Anfang in der Liste zu geben. Der Einstieg in eine zyklisch verkettete lineare Liste kann deshalb an jedem beliebigen Element der Liste erfolgen.

Will man von einem Element einer einfach verketteten linearen Liste zum Vorgänger dieses Elements gelangen, so ist dies nur mit großem Aufwand möglich. Man schafft in diesem Falle Abhilfe durch eine weitere Verzeigerung zum Vorgänger; jedes Element besitzt also zwei Zeiger. Eine solche Liste nennt man *doppelt verkettete lineare Liste.* Wenn wir bei einer solchen Liste nicht nur einen Zeiger auf das erste, sondern auch einen Zeiger auf das letzte Element der Liste einrichten, sind die Interpretationen »Anfang« und »Ende« beliebig austauschbar.

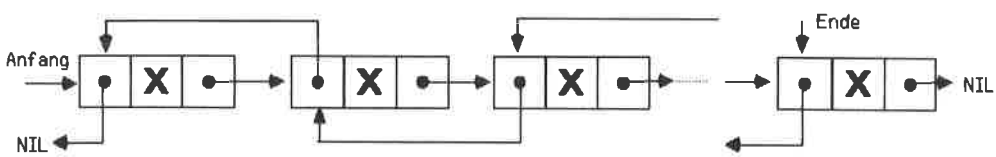


Bild 23.4. *Darstellung einer doppelt verketteten linearen Liste*

Wir erweitern obiges Beispiel einer einfach verketteten linearen Liste zu einer doppelt verketteten linearen Liste:

```
ANFANG:  DEFW    ELEM1    ; Zeiger auf erstes Element
ENDE:    DEFW    ELEM3    ; Zeiger auf letztes Element
ELEM2:   DEFW    ELEM1    ; Zeiger auf erstes Element
         DEFB    'B'      ; Wert des zweiten Elements
```

```
        DEFW    ELEM3       ; Zeiger auf drittes Element
ELEM3:  DEFW    ELEM2       ; Zeiger auf zweites Element
        DEFB    'C'         ; Wert des dritten Elements
        DEFW    NIL         ; Zeiger ins Leere
ELEM1:  DEFW    NIL         ; Zeiger ins Leere
        DEFB    'A'         ; Wert des ersten Elements
        DEFW    ELEM2       ; Zeiger auf zweites Element
```

Mittels des Unterprogramms VORG gelangen wir von einem Element der Liste zu seinem Vorgänger; das Null-Flag wird gesetzt, falls kein Vorgänger existiert:

```
VORG:   CALL    ISTNIL      ; Zeiger auf NIL testen
        RET     Z           ; ungueltiger Zeiger
        LD      E,(HL)      ; Adresse des
        INC     HL          ; Vorgaengers
        LD      D,(HL)      ; holen
        DEC     HL          ; Zeiger restaurieren
        RET
```

Das Unterprogramm NACHFO müssen wir geringfügig abändern:

```
NACHF:  CALL    ISTNIL      ; Zeiger auf NIL testen
        RET     Z           ; ungueltiger Zeiger
        PUSH    HL          ; Zeiger sichern
        INC     HL          ; auf Adresse
        INC     HL          ; des Nachfolgers
        INC     HL          ; zeigen
        LD      E,(HL)      ; Adresse des
        INC     HL          ; Nachfolgers
        LD      D,(HL)      ; holen
        POP     HL          ; Zeiger restaurieren
        RET
```

Eine Anwendung der doppelt verketteten linearen Listen werden wir im Unterkapitel »Darstellung von Puffern durch Listen« kennenlernen.

Übungen

1. Schreibe ein Unterprogramm, das zwei Zeiger auf je eine einfach verkettete lineare Liste erhält und die beiden Listen zu einer Liste vereinigt (Anhängen der einen Liste an das letzte Element der anderen Liste).

2. Was ändert sich im Unterprogramm NACHFO, wenn die Werte der Elemente nicht vom Typ »Byte« sind?

3. Schreibe ein Unterprogramm, das zum Zeiger auf ein Element einer zyklisch verketteten linearen Liste den Zeiger auf den Nachfolger dieses Elements liefert.

23.2 Darstellung von Mengen durch Listen

Bei der Suche nach der Repräsentation einer Menge bietet sich die einfach verkettete lineare Liste an. Aus Effizienzgründen soll jedes Element der Menge in der Liste genau einmal aufgeführt sein (siehe Kapitel »Mengen«). Haben wir es mit einer Menge mit Ordnungsrelation zu tun, so können wir die Ordnung auf die Repräsentation übertragen (und diese so zu einer geordneten Repräsentation machen), indem wir ein Mengenelement genau dann vor ein anderes Mengenelement in der Liste setzen, wenn jenes Element in der Ordnung der Menge vor dem anderen steht. Wir wollen im folgenden einige Operationen auf einer geordneten Repräsentation einer Zeichenmenge realisieren.

Die Kardinalität der Menge ist die Anzahl der Listenelemente (die Länge der Liste). Diese gewinnen wir dadurch, daß wir - ausgehend vom Einstieg in die Liste - von einem Element zum nächsten übergehen, bis das Ende der Liste erreicht ist, und dabei einen Zähler mitführen. Folgende Routine liefert die Kardinalität im B-Register, wenn ihr im HL-Register der Einstieg in die Liste übergeben wird:

```
KARD:   LD      B,0         ; Zaehler initialisieren
        PUSH    DE          ; Registerinhalte
        PUSH    HL          ; sichern
SUCHE:  CALL    NACHFO      ; Zeiger auf Nachfolger bestimmen
        JP      Z,FERTIG    ; kein Nachfolger vorhanden
        INC     B           ; Element zaehlen
        EX      DE,HL       ; Zeiger auf Nachfolger zu
                            ; Zeiger auf Element machen
        JP      SUCHE       ; gesamte Liste abarbeiten
FERTIG: POP     HL          ; Register
        POP     DE          ; restaurieren
        RET
```

Als nächstes wollen wir feststellen, ob ein bestimmtes Element in der Menge enthalten ist; dies soll durch Setzen des Null-Flags angezeigt werden. Das Zeichen übergeben wir im A-Register. Die Ordnung der Repräsentation nehmen wir aufsteigend an, das heißt, daß der Wert des Nachfolgers eines Elements stets größer ist als der Wert des Elements selbst. Wir können die Suche dann abbrechen, wenn wir den gesuchten Wert gefunden haben, oder aber wenn der Wert des zuletzt untersuchten Elements größer als der Vergleichs-Wert ist (alle folgenden Elemente

besitzen ja noch größere Werte). Der zweite Fall stellt normalerweise eine erhebliche Laufzeitverkürzung dar; dies ist der spezielle Vorteil geordneter Repräsentationen.

```
ISTIN:   PUSH   DE        ; Registerinhalte
         PUSH   HL        ; sichern
         CALL   ISTNIL    ; Zeiger auf NIL testen
SUCHE:   JP     Z,ENDE    ; Ende der Liste erreicht
         CP     (HL)      ; Wert des Elements mit
                          ; Vergleichs-Wert vergleichen
         JP     Z,FERTIG  ; Wert gefunden
         JP     C,FERTIG  ; Wert nicht in der Liste
         CALL   NACHFO    ; Adresse des Nachfolgers holen
         EX     DE,HL     ; auf Nachfolger zeigen
         JP     SUCHE     ; gegebenenfalls gesamte Liste
                          ; durchgehen
ENDE:    LD     L,0       ; Null-Flag
         INC    L         ; loeschen
FERTIG:  POP    HL        ; Register
         POP    DE        ; restaurieren
         RET
```

Eine weitere notwendige Operation ist das Hinzufügen eines Elements zu der Menge; dabei müssen wir beachten, daß das Element nicht versehentlich zweimal in der Liste auftaucht. Die Ordnung der Repräsentation soll durch einen Einfügevorgang natürlich nicht zerstört werden. Das neue Element wird bereits in einem geeigneten Verbund bereitgestellt; ein Zeiger auf diesen Verbund wird im DE-Register übergeben. Zurückgeliefert wird ein Zeiger auf die aktualisierte Liste (im HL-Register):

```
HINEIN:  LD     A,(DE)    ; Wert des einzufuegenden
                          ; Elements holen
         CALL   ISTNIL    ; Zeiger auf NIL testen
         JP     Z,VORNE   ; leere Liste,
                          ; Element vorne einfuegen
         CP     (HL)      ; Wert des ersten Elements der
                          ; Liste mit Wert des
                          ; einzufuegenden Elements
                          ; vergleichen
         RET    Z         ; Element schon in der Liste
         JP     NC,SUCHE  ; Wert des ersten Elements der
                          ; Liste kleiner als Wert des
                          ; einzufuegenden Elements
VORNE:   EX     DE,HL     ; Zeiger tauschen
         INC    HL        ; auf Zeiger des Elements zeigen
```

```
        LD      (HL),E      ; alte Liste
        INC     HL          ; an neues Element
        LD      (HL),D      ; anfuegen
        DEC     HL          ; Zeiger auf Anfang der neuen
        DEC     HL          ; Liste generieren
        RET
SUCHE:  PUSH    HL          ; Zeiger auf Anfang der Liste
                            ; sichern
        PUSH    DE          ; Zeiger auf einzufuegendes
                            ; Element sichern
SUCHEN: CALL    NACHFO      ; Zeiger auf Nachfolger holen
        EX      DE,HL       ; Zeiger tauschen
        CALL    ISTNIL      ; auf Ende der Liste testen
        JP      Z,EINF      ; Ende der Liste erreicht,
                            ; Element hinten anfuegen
        CP      (HL)        ; Wert des Elements mit Wert des
                            ; einzufuegenden Elements
                            ; vergleichen
        JP      Z,FERTIG    ; Element schon in der Liste
        JP      NC,SUCHEN   ; Wert des Elements immer noch
                            ; kleiner als Wert des
                            ; einzufuegenden Elements
EINF:   EX      (SP),HL     ; Zeiger auf einzufuegendes
                            ; Element holen, Zeiger auf Rest
                            ; der Liste auf dem
                            ; Stapel ablegen
        EX      DE,HL       ; Zeiger tauschen
        INC     HL          ; neues
        LD      (HL),E      ; Element an
        INC     HL          ; Anfang der Liste
        LD      (HL),D      ; anfuegen
        EX      DE,HL       ; Zeiger tauschen
        POP     DE          ; Zeiger auf Rest der Liste holen
        INC     HL          ; Rest der Liste
        LD      (HL),E      ; an neues
        INC     HL          ; Element
        LD      (HL),D      ; anfuegen
        POP     HL          ; Zeiger auf Anfang
                            ; der Liste holen
        RET
FERTIG: POP     DE          ; Register
        POP     HL          ; restaurieren
        RET
```

Als letzte grundlegende Mengenoperation beschreiben wir das Entfernen eines Elements aus der Liste. Der Wert, der entfernt werden soll, steht dabei im A-Register. Das HL-Register enthält zu Beginn einen Zeiger auf den Anfang der Liste, nach Beendigung der Operation einen Zeiger auf den Anfang der aktualisierten Liste.

```
LOESCH:   CALL    ISTNIL      ; auf leere Liste testen
          RET     Z           ; leere Liste,
                              ; nichts zu entfernen
          CP      (HL)        ; Wert des Elements mit zu
                              ; entfernendem Wert vergleichen
          RET     C           ; Element nicht in der Liste,
                              ; nichts zu tun
          JP      NZ,SUCHE    ; Werte stimmen nicht ueberein
          INC     HL          ; Zeiger
          LD      E,(HL)      ; des
          INC     HL          ; Nachfolgers
          LD      D,(HL)      ; bestimmen
          EX      DE,HL       ; Zeiger tauschen
          RET
SUCHE:    PUSH    HL          ; Zeiger auf Anfang der Liste
                              ; sichern
SUCHEN:   CALL    NACHFO      ; Zeiger auf Nachfolger holen
          JP      Z,FERTIG    ; Liste zu Ende,
                              ; nichts zu entfernen
          EX      DE,HL       ; Zeiger tauschen
          CP      (HL)        ; Wert des Elements mit zu
                              ; entfernendem Wert vergleichen
          JP      Z,ENTFER    ; Werte stimmen ueberein
          JP      NC,SUCHEN   ; weitersuchen
FERTIG:   POP     HL          ; Zeiger auf Anfang der Liste
                              ; restaurieren
          RET
ENTFER:   INC     HL          ; Zeiger auf
          LD      C,(HL)      ; Nachfolger des
          INC     HL          ; zu entfernenden
          LD      B,(HL)      ; Elements holen
          EX      DE,HL       ; Zeiger tauschen
          INC     HL          ; Zeiger auf Nachfolger des
          LD      (HL),C      ; zu entfernenden Elements
          INC     HL          ; in Vorgaenger des zu
          LD      (HL),B      ; entfernenden Elements kopieren
          POP     HL          ; Zeiger auf Anfang der Liste
                              ; restaurieren
          RET
```

Übungen

1. Schreibe ein Unterprogramm, das die Vereinigung zweier Mengen bildet.

2. Schreibe ein Unterprogramm, das die Differenz zweier Mengen bildet.

3. Schreibe ein Unterprogramm, das den Schnitt zweier Mengen bildet.

4. Schreibe ein Unterprogramm, das feststellt, ob eine Menge in einer anderen Menge enthalten ist.

23.3 Darstellung von Stapeln durch Listen

Bei echten Stapeln ist immer nur das oberste Stapel-Element erreichbar; erst durch Wegnehmen des bisher obersten Stapel-Elements kommt das darunterliegende Stapel-Element zum Vorschein (falls der Stapel nicht leer ist). Die geeignete Datenstruktur zur Darstellung eines Stapels ist damit die einfach verkettete lineare Liste; das erste Listenelement stellt das oberste Stapel-Element dar, der Einstieg in die Liste stellt den Stapel-Zeiger dar.

Das schwierigste Problem beim gleichzeitigen Betreiben mehrerer Stapel dieser Art ist die Beschaffung von Speicherplatz für die Stapel-Elemente, die neu zu einem der Stapel hinzukommen beziehungsweise die Wiederverwendung freigegebenen Speicherplatzes. Wir sehen uns deshalb zunächst eine recht gebräuchliche Form von Speicherverwaltung für derartige Probleme an.

Wir reservieren einen Teil des Speichers zum ausschließlichen Gebrauch durch die Stapel. Dieser Speicherbereich soll ab der Adresse SPEICH beginnen; die Anzahl der Elemente, die in diesem Speicherbereich Platz finden, soll durch ANZAHL bezeichnet werden, die Anzahl der Bytes, die der Wert eines Elements benötigt, durch LAENGE. Der gesamte Speicherbereich ist damit ANZAHL * (LAENGE + 2) Bytes lang, da zum Wert jeden Elements noch der Zeiger auf den Nachfolger des Elements tritt.

Als erstes *formatieren* wir den freien Speicher und bringen ihn damit in Form einer einfach verketteten linearen Liste, der sogenannten *Freiliste.* Die Freiliste enthält alle Elemente des Speicherbereichs, die zur Aufnahme eines neuen Werts und zum Einfügen in einen der Stapel benutzt werden können. Der Einstieg in die Freiliste wird durch einen Zeiger gegeben, der in der Variablen FREILI abgespeichert wird. Die Formatierung geschieht folgendermaßen:

```
FORMAT:   LD      HL,SPEICH      ; Zeiger auf freien
                                 ; Speicherbereich laden
          LD      (FREILI),HL    ; Zeiger auf Freiliste
                                 ; abspeichern
          LD      BC,ANZAHL-1    ; Anzahl der Elemente der
                                 ; Freiliste, die einen Nachfolger
                                 ; besitzen, laden
```

```
VERKET:  LD      DE,LAENGE   ; Laenge des Werts eines
                             ; Elements laden
         ADD     HL,DE       ; Adresse des Zeigers auf
                             ; Nachfolger berechnen
         EX      DE,HL       ; Adresse sichern
         LD      HL,2        ; Zeiger auf
         ADD     HL,DE       ; Nachfolger berechnen
         EX      DE,HL       ; Zeiger tauschen
         LD      (HL),E      ; Zeiger auf Nachfolger
         INC     HL          ; in Element
         LD      (HL),D      ; eintragen
         EX      DE,HL       ; Zeiger tauschen
         DEC     BC          ; Zaehler vermindern
         LD      A,B         ; Zaehler auf
         OR      C           ; Null testen
         JP      NZ,VERKET   ; Verkettung fortsetzen
         LD      DE,LAENGE   ; Laenge des Werts eines
                             ; Elements laden
         ADD     HL,DE       ; Adresse des Zeigers berechnen
         LD      DE,NIL      ; NIL laden
         LD      (HL),E      ; letzter Zeiger
         INC     HL          ; der Liste zeigt
         LD      (HL),D      ; ins Leere
         RET
```

Die Stapel-Zeiger unserer Stapel tragen zunächst alle den Wert NIL.

Wir wollen nun die Operation PUSH auf einem unserer Stapel realisieren. Das HL-Register enthält die Adresse des zugehörigen Stapel-Zeigers. Der zu sichernde Wert wird irgendwo im Speicher bereitgestellt; die Routine erhält einen Zeiger auf diesen Wert (im DE-Register). Das Sichern geschieht in mehreren Phasen:

1. Auf Stapelüberlauf prüfen: Ist die Freiliste leer, so kann kein Speicherplatz für das neue Element beschafft werden; in diesem Fall soll das Null-Flag gesetzt und die Operation abgebrochen werden.
2. Sichern: Kopieren des Werts in das erste Element der Freiliste.
3. Speicherplatz vergeben: Entnehmen des ersten Elements der Freiliste und Einfügen als erstes Element des Stapels. Beides wird durch Umhängen von Zeigern realisiert.

Die Routine lautet damit:

```
PUSH:    PUSH    HL          ; Adresse des Stapel-Zeigers
                             ; sichern
         LD      HL,(FREILI) ; Zeiger auf Freiliste holen
```

```
        CALL    ISTNIL      ; auf Stapelueberlauf testen
        JP      NZ,WEITER   ; kein Stapelueberlauf
        POP     HL          ; Adresse des Stapel-Zeigers
                            ; restaurieren
        RET
WEITER: PUSH    HL          ; Zeiger auf neues Element
                            ; sichern
        LD      BC,LAENGE   ; Laenge des Werts
                            ; eines Elements laden
        LDIR                ; Wert in neues Element kopieren
        LD      E,(HL)      ; Zeiger auf Rest
        INC     HL          ; der Freiliste
        LD      D,(HL)      ; holen
        LD      (FREILI),DE ; neuen Zeiger auf Freiliste
                            ; eintragen
        POP     BC          ; Zeiger auf neues Element holen
        EX      DE,HL       ; Adresse des Stapel-Zeigers
        POP     HL          ; holen
        INC     HL          ; neues Element in Stapel
                            ; einhaengen
        LD      A,(HL)
        LD      (DE),A
        LD      (HL),B
        DEC     DE
        DEC     HL
        LD      A,(HL)
        LD      (DE),A
        LD      (HL),C
        RET
```

Das Holen eines Elements vom Stapel erfolgt ebenfalls in mehreren Phasen; bei erfolgreicher Durchführung der Operation wird der Wert des obersten Stapel-Elements in einen Speicherbereich kopiert, auf den das DE-Register zeigt. Die einzelnen Phasen sind:

1. Prüfen auf Stapelunterlauf: Ist der Wert des Stapel-Zeigers gleich NIL, so wird die Routine abgebrochen und das Null-Flag gesetzt.
2. Holen: Kopieren des Werts des obersten Stapel-Elements in einen vorgegebenen Speicherbereich.
3. Entfernen des obersten Stapel-Elements und Einfügen in Freiliste durch Umhängen von Zeigern (Speicherbereinigung, engl. garbage collection).

```
POP:    PUSH    HL          ; Adresse des Stapel-Zeigers
                            ; sichern
        PUSH    DE          ; Zeiger auf Ablageort sichern
```

```
LD      E,(HL)      ; Stapel-
INC     HL          ; Zeiger
LD      D,(HL)      ; holen
EX      DE,HL       ; auf Stapelunterlauf
CALL    ISTNIL      ; testen
POP     DE          ; Stapel
POP     BC          ; korrigieren
RET     Z           ; Stapelunterlauf
PUSH    BC          ; Adresse des Stapel-Zeigers
                    ; sichern
LD      BC,LAENGE   ; Laenge des Werts eines Elements
LDIR                ; Wert kopieren
LD      BC,(FREILI) ; Zeiger auf Freiliste holen
LD      E,(HL)      ; neuen
LD      (HL),C      ; Wert
INC     HL          ; des
LD      D,(HL)      ; Stapel-Zeigers
LD      (HL),B      ; berechnen
POP     HL          ; Adresse des Stapel-Zeigers
                    ; holen
LD      C,(HL)      ; bisher
LD      (HL),E      ; oberstes
INC     HL          ; Element
LD      B,(HL)      ; des Stapels
LD      (HL),D      ; in Freiliste
LD      (FREILI),BC ; einhaengen
RET
```

Übungen

1. Optimiere die Routine PUSH und POP für LAENGE = 1 (Stapel von Bytes).

23.4 Darstellung von Puffern durch Listen

Da Produzent und Konsument den Puffer an unterschiedlichen Stellen bearbeiten, ist für Puffer als Datenstruktur eine einfach verkettete lineare Liste mit zwei Zeigern angemessen. Der Produzent erweitert den Puffer am Listenende, der Konsument entnimmt ihm am Listenanfang Elemente. Analog zu den Operationen PUSH und POP auf Stapeln definieren wir die Operationen FUELLE des Produzenten und LEERE des Konsumenten. FUELLE bricht mit gesetztem Null-Flag ab, wenn kein weiterer Speicherplatz zugeteilt werden kann, LEERE bricht mit gesetztem Null-Flag ab, wenn der Puffer leer ist. Wir verwenden wieder die Speicher-

verwaltung aus dem vorangegangenen Unterkapitel und auch alle übrigen Rahmenbedingungen (statt je eines Stapel-Zeigers haben wir hier allerdings je einen Produzenten-Zeiger und je einen Konsumenten-Zeiger).

Alle Produzenten- und Konsumenten-Zeiger tragen zunächst den Wert NIL.

Die Unterprogramme FUELLE und LEERE sehen folgendermaßen aus (HL zeigt auf den Produzenten-Zeiger, DE auf den Konsumenten-Zeiger, BC auf den Ablageort des Werts):

```
FUELLE:  EX      DE,HL        ; Adresse des Konsumenten-
         PUSH    HL           ; Zeigers sichern
         LD      HL,(FREILI)  ; Freilisten-Zeiger holen
         CALL    ISTNIL       ; und auf NIL testen
         JP      NZ,KEINUE    ; kein Pufferueberlauf
         POP     HL           ; Stapel korrigieren
         RET                  ; Ende mit Fehler
KEINUE:  PUSH    HL           ; Freilisten-Zeiger sichern
         PUSH    DE           ; Adresse des Produzenten-
                              ; Zeigers sichern
         LD      D,B          ; Wert
         LD      E,C          ; in
         EX      DE,HL        ; neuem
         LD      BC,LAENGE    ; Pufferelement
         LDIR                 ; ablegen
         EX      DE,HL        ; neuen
         D       DE,NIL       ; Freilisten-
         LD      C,(HL)       ; Zeiger
         LD      (HL),D       ; berechnen,
         INC     HL           ; letztes
         LD      B,(HL)       ; Pufferelement
         LD      (HL),E       ; markieren
         LD      (FREILI),BC  ; neuen Freilisten-Zeiger
                              ; eintragen
         POP     HL           ; Adresse des Produzenten-
                              ; Zeigers holen
         POP     BC           ; alten Freilisten-Zeiger holen
         LD      E,(HL)       ; neuen Wert
         LD      (HL),C       ; des Produzenten-Zeigers
         INC     HL           ; eintragen,
         LD      D,(HL)       ; alten Wert des
         LD      (HL),B       ; Produzenten-Zeigers holen
         EX      DE,HL        ; auf vorher leeren Puffer
         CALL    ISTNIL       ; testen
         JP      Z,LEER       ; Puffer war bisher leer
         LD      DE,LAENGE    ; Zeiger auf
```

```
        ADD     HL,DE       ; Verweis berechnen
WEITER: LD      (HL),C      ; Liste schliessen
        INC     HL          ; beziehungsweise
        LD      (HL),B      ; neuen Konsumenten-Zeiger
                            ; eintragen
        POP     HL          ; Stapel korrigieren
        RET
LEER:   EX      (SP),HL     ; Adresse des Konsumenten-
                            ; Zeigers holen
        JP      WEIT        ; neuen Konsumenten-Zeiger
                            ; eintragen
LEERE:  EX      DE,HL       ; Adresse des Produzenten-
        PUSH    DE          ; Zeigers sichern
        LD      E,(HL)      ; Konsumenten-
        INC     HL          ; Zeiger
        LD      D,(HL)      ; holen
        EX      DE,HL       ; Konsumenten-Zeiger
        CALL    ISTNIL      ; auf NIL testen
        JP      NZ,KUNTER   ; kein Pufferunterlauf
        POP     HL          ; Adresse des Produzenten-
                            ; Zeigers restaurieren
        RET                 ; Ende mit Fehler
KUNTER: PUSH    DE          ; Adresse des Konsumenten-
                            ; Zeigers sichern
        PUSH    HL          ; neuen Freilisten-Zeiger sichern
        LD      D,B         ; Wert
        LD      E,C         ; aus dem
        LD      BC,LAENGE   ; Puffer
        LDIR                ; entnehmen
        LD      BC,(FREILI) ; alten Freilisten-Zeiger holen
        POP     DE          ; neuen Freilisten-Zeiger
        LD      (FREILI),DE ; eintragen
        LD      E,(HL)      ; Freiliste
        LD      (HL),C      ; vervollstaendigen,
        INC     HL          ; neuen
        LD      D,(HL)      ; Konsumenten-Zeiger
        LD      (HL),B      ; holen
        POP     HL          ; Adresse des Konsumenten-
                            ; Zeigers holen
        LD      (HL),D      ; neuen
        DEC     HL          ; Konsumenten-Zeiger
        LD      (HL),E      ; eintragen
        POP     HL          ; Adresse des Produzenten-
```

```
                                ; Zeigers holen
EX          DE,HL               ; auf leeren
CALL        ISTNIL              ; Puffer testen
RET         NZ                  ; Puffer nicht leer
EX          DE,HL               ; neuen
LD          (HL),E              ; Produzenten-
INC         HL                  ; Zeiger
LD          (HL),D              ; eintragen
XOR         A                   ; Null-Flag
INC         A                   ; loeschen
RET
```

Übungen

1. Optimiere die Routine FUELLE und LEERE für LAENGE = 1 (Puffer von Bytes).

23.5 Bäume

Ein *Baum* ist eine verzeigerte Datenstruktur, in der jedes Element eine beliebige Anzahl von Nachfolgern besitzt. Die Elemente eines Baums nennt man *Knoten*. Genau ein Knoten eines nichtleeren Baums besitzt keinen Vorgänger, dieser heißt *Wurzel*; alle anderen Knoten des Baums besitzen genau einen Vorgänger. Die Verzeigerung zum Vorgänger braucht nicht explizit vorhanden zu sein, sie ergibt sich aus der Verzeigerung der Knoten mit ihren Nachfolgern. Entfernt man aus einem Baum die Wurzel mit ihren Zeigern, so zerfällt der Baum in Teilbäume, deren Wurzeln genau die Nachfolger der ursprünglichen Wurzel sind. Zu jedem Knoten des Baums kann man von der Wurzel aus auf genau eine Weise mittels der Nachfolger-Verweise gelangen. Einige Beispiele für Bäume sind:

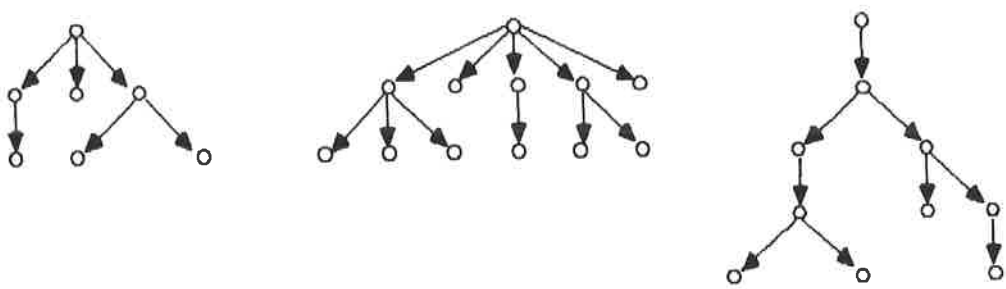


Bild 23.5. *Beispiele von Bäumen*

Manchmal legt man auch die Zahl der Nachfolger fest; die gebräuchlichste Form ist der sogenannte *Binärbaum*, in dem jeder Knoten zwei Nachfolger hat. Die Nachfolger können leere Bäume sein; ein Zeiger auf einen leeren Baum trägt den Wert NIL.

Ist die Zahl der Nachfolger eines Knotens beliebig, so stellt man die Knoten am besten durch einfach verkettete lineare Listen dar, deren Elemente Zeiger als Werte besitzen. Ein mögliches Schema wäre folgendes: Der Wert des ersten Listenelements ist ein Zeiger auf den Wert des Knotens; besitzen die Knoten keine Werte, so entfällt dieses Element. Der Wert des zweiten Listenelements ist ein Zeiger auf den Vorgänger des Knotens beziehungsweise NIL, falls der Knoten die Wurzel ist; wenn kein Verweis auf den Vorgänger gewünscht wird, entfällt dieses Element. Die Werte der restlichen Elemente der Liste stellen die Zeiger auf die Nachfolger des Knotens dar (siehe dazu die Abbildung).

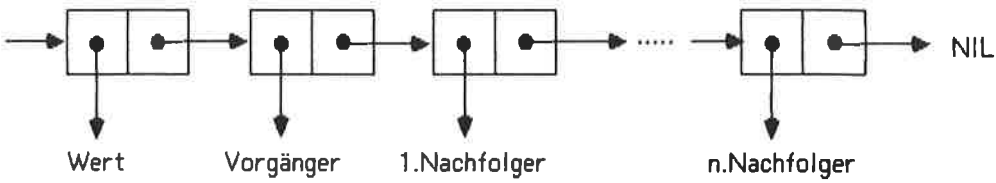


Bild 23.6. *Darstellung eines Knotens als Liste*

Das Verfahren ist ziemlich aufwendig, aber für alle Formen von Bäumen anwendbar. Die Bearbeitung eines solchen Baums erfolgt mit den Techniken für lineare Listen.

Die Nachfolger eines Knotens sind in der Repräsentation linear geordnet; dies ist bei Arboreszenzen im Sinne der Graphentheorie nicht a priori der Fall.

Bei Bäumen, deren Knoten eine feste Zahl von Nachfolgern besitzen, stellt man die Knoten meist durch Verbunde dar; die Komponenten eines solchen Verbunds sind der Wert des Knotens (falls existent), der Zeiger auf den Vorgänger des Knotens (falls existent) und die Zeiger auf die Nachfolger des Knotens. Wir bringen als Beispiel einen (kleinen) Binärbaum (mit Vorgänger-Verweisen); die Werte der Knoten sind Bytes:

```
BAUM:   DEFW    WURZEL      ; Verweis auf die Wurzel
WURZEL: DEFB    12          ; Wert des Knotens
        DEFW    NIL         ; Vorgaenger
        DEFW    ELEM1       ; linker Nachfolger
        DEFW    ELEM2       ; rechter Nachfolger
ELEM1:  DEFB    34          ; Wert des Knotens
        DEFW    WURZEL      ; Vorgaenger
        DEFW    ELEM3       ; linker Nachfolger
        DEFW    ELEM4       ; rechter Nachfolger
ELEM2:  DEFB    21          ; Wert des Knotens
        DEFW    WURZEL      ; Vorgaenger
        DEFW    NIL         ; linker Nachfolger
        DEFW    ELEM5       ; rechter Nachfolger
ELEM3:  DEFB    7           ; Wert des Knotens
        DEFW    ELEM1       ; Vorgaenger
        DEFW    ELEM6       ; linker Nachfolger
```

```
        DEFW    NIL     ; rechter Nachfolger
ELEM4:  DEFB    56      ; Wert des Knotens
        DEFW    ELEM1   ; Vorgaenger
        DEFW    NIL     ; linker Nachfolger
        DEFW    NIL     ; rechter Nachfolger
ELEM5:  DEFB    29      ; Wert des Knotens
        DEFW    ELEM2   ; Vorgaenger
        DEFW    NIL     ; linker Nachfolger
        DEFW    NIL     ; rechter Nachfolger
ELEM6:  DEFB    44      ; Wert des Knotens
        DEFW    ELEM3   ; Vorgaenger
        DEFW    NIL     ; linker Nachfolger
        DEFW    NIL     ; rechter Nachfolger
```

Die Knoten könnten natürlich auch über den ganzen Speicher verstreut sein. Dieser Baum sieht folgendermaßen aus:

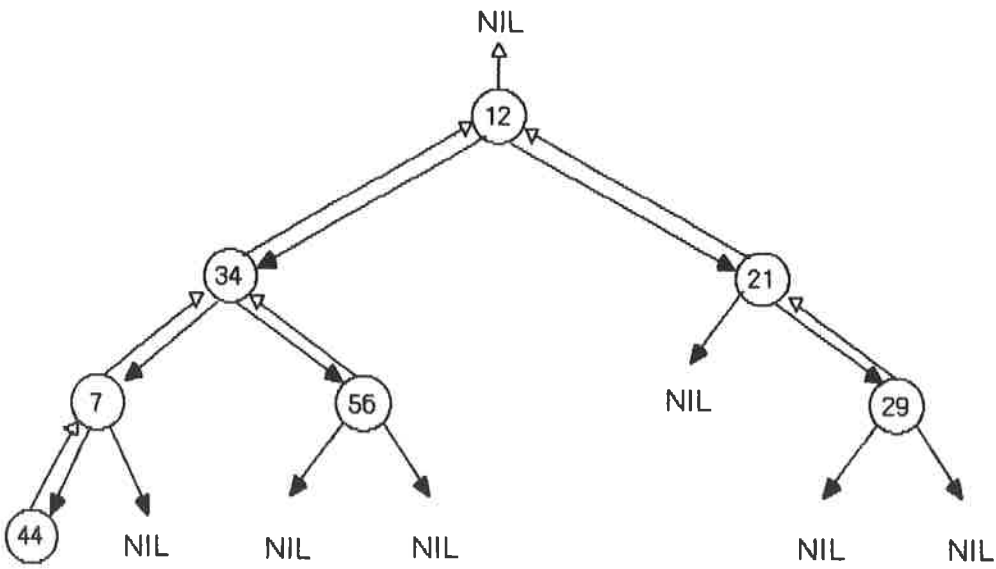


Bild 23.7. *Beispiel eines Binärbaums*

Bäume sind von ihrem Bildungsschema her rekursive Datenstrukturen: Ein Baum ist entweder ein leerer Baum oder er hat eine Wurzel, an der wiederum Bäume hängen. Deshalb werden die meisten Operationen auf Bäumen rekursiv ausgeführt. Wir zeigen als Beispiel eine Funktion auf Binärbäumen (ohne Vorgängerverweise), deren Elemente Werte vom Typ »Byte« tragen (siehe obiges Beispiel), die zu einem vorgelegten Wert und einem Baum entscheidet, ob der Wert im Baum vorkommt (Null-Flag wird gesetzt) oder nicht. Der Wert wird im A-Register übergeben, der Zeiger auf die Wurzel des Baums im HL-Register.

```
INBAUM:  CALL    ISTNIL     ; auf leeren Baum testen
         JP      NZ,NLEER   ; Baum nicht leer
         LD      E,0        ; Null-Flag
         INC     E          ; loeschen
         RET                ; Wert nicht im Baum
NLEER:   CP      (HL)       ; vorgelegten Wert mit Wert
                            ; der Wurzel vergleichen
         RET     Z          ; Werte stimmen ueberein,
                            ; Wert im Baum enthalten
         INC     HL         ; Zeiger auf
         LD      E,(HL)     ; linken
         INC     HL         ; Nachfolger
         LD      D,(HL)     ; der Wurzel
         EX      DE,HL      ; holen
         PUSH    DE         ; Zeiger auf Wurzel sichern
         CALL    INBAUM     ; Test fuer linken Teilbaum
                            ; durchfuehren
         POP     HL         ; Zeiger auf Wurzel restaurieren
         RET     Z          ; Suche im linken Teilbaum
                            ; erfolgreich,
                            ; Wert im Baum enthalten
         INC     HL         ; Zeiger auf
         LD      E,(HL)     ; rechten
         INC     HL         ; Nachfolger
         LD      D,(HL)     ; der Wurzel
         EX      DE,HL      ; holen
         CALL    INBAUM     ; Test fuer rechten Teilbaum
                            ; durchfuehren
         RET
```

Übungen

1. Schreibe ein Unterprogramm, das feststellt, ob zwei Binärbäume übereinstimmen.

23.6 Graphen

Der Vollständigkeit halber geben wir noch Darstellungsmöglichkeiten für Graphen an. Ein *Graph* ist die allgemeinste Art von verzeigerter Datenstruktur; er besteht aus *Knoten* (den Elementen des Graphen) und *Kanten* (Bezügen zwischen den Knoten). Die Bezüge geben an, ob man von einem Knoten zum anderen gelangen kann. Jede Kante kann mit einem Wert belegt sein; jeder Knoten kann ebenfalls mit einem Wert belegt sein. In folgender Abbildung werden einige Graphen gezeigt:

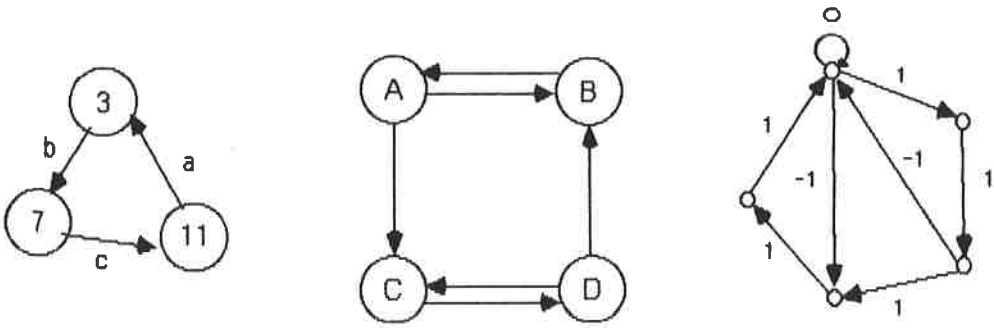


Bild 23.8. *Beispiele für Graphen*

Die eingeführte Form von Graph entspricht – strenggenommen – gerichteten Graphen, bei denen es auf die Richtung des Übergangs von einem Knoten zum anderen wesentlich ankommt. Ungerichtete Graphen geben dagegen nur an, ob zwei Knoten verbunden sind. Wir können aber jeden ungerichteten Graphen in einen gerichteten Graphen umwandeln, indem wir zwei durch eine Kante verbundene Knoten auch durch eine Kante in umgekehrter Richtung mit gleichem Wert verbinden. Ein Beispiel für einen ungerichteten Graphen und seine Transformation in einen gerichteten Graphen wäre:

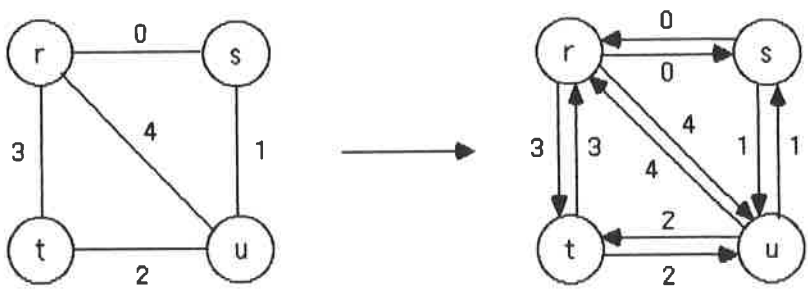


Bild 23.9. *Umwandlung eines ungerichteten Graphen in einen gerichteten Graphen*

Zur Darstellung der Knoten könnten wir im Prinzip wieder wie bei den Bäumen vorgehen und lineare Listen verwenden; während es aber für Bäume mit fester Nachfolgerzahl einen Sinn ergab, Zeiger mit Wert NIL zuzulassen, kommt dies bei Graphen nicht vor, da immer nur Knoten miteinander verbunden sind. Wir stellen deshalb einen Knoten durch einen Verbund dar, der alle Zeiger auf andere Knoten sowie die Werte der Kanten und des Knotens selbst enthält; die einzelnen Verbunde können in der Länge variieren, weshalb wir als Ende-Markierung einen Zeiger mit Wert NIL anfügen. Wir stellen den im folgenden Bild gezeigten Graphen in dieser Form dar:

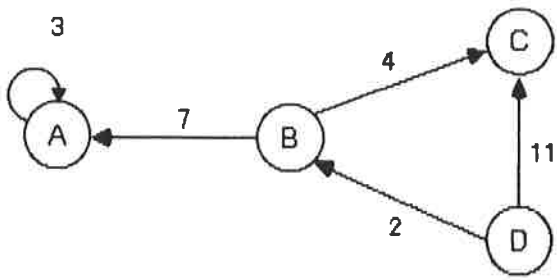


Bild 23.10. *Beispiel eines gerichteten Graphen*

```
KNOT1:  DEFB    'A'     ; Wert des Knotens
        DEFW    KNOT1   ; Kante
        DEFB    3       ; Wert der Kante
        DEFW    NIL     ; Ende-Markierung
KNOT2:  DEFB    'B'     ; Wert des Knotens
        DEFW    KNOT1   ; Kante
        DEFB    7       ; Wert der Kante
        DEFW    KNOT3   ; Kante
        DEFB    4       ; Wert der Kante
        DEFW    NIL     ; Ende-Markierung
KNOT3:  DEFB    'C'     ; Wert des Knotens
        DEFW    NIL     ; Ende-Markierung
KNOT4:  DEFB    'D'     ; Wert des Knotens
        DEFW    KNOT2   ; Kante
        DEFB    2       ; Wert der Kante
        DEFW    KNOT3   ; Kante
        DEFB    11      ; Wert der Kante
        DEFW    NIL     ; Ende-Markierung
```

Die Algorithmen, in denen Graphen verwendet werden, sind meist recht kompliziert; wir verzichten deshalb hier auf Beispiele.

Wie bei Bäumen die Nachfolger, so sind bei den Graphen die Nachbarn eines Knotens in der Repräsentation linear geordnet, was bei Graphen im Sinne der Graphentheorie nicht a priori gilt.

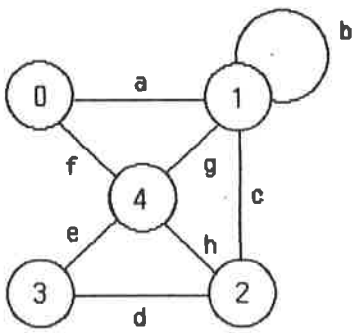
Graphen kommen vorwiegend in dem wichtigen Gebiet der mathematischen Optimierung zum Einsatz. Beispiele sind:

- Weg-Minimierung beim Transport von Gütern per LKW (ungerichteter Graph, die anzufahrenden Stationen stellen die Knoten dar, die Straßenverbindungen die Kanten, die Entfernungen die Werte der Kanten)
- Weg-Minimierung bei der städtischen Müllabfuhr oder bei Briefträgern (gerichteter Graph wegen der Einbahnstraßen)

- Durchsatz-Maximierung in der Fertigung von Waren (gerichteter Graph). Mittels Graphen lassen sich auch Zustandsübergänge von Automaten oder zeitliche und logische Abhängigkeiten beliebiger Vorgänge modellieren.

Übungen

1. Implementiere folgenden ungerichteten Graphen:



24
Ganze Zahlen

Wir haben bereits mit ganzen Zahlen gearbeitet, genauer mit binär-codierten ganzen Zahlen: mit Bytes und Worten. Diesen Ansatz kann man in zwei Richtungen verallgemeinern: Manchmal ist der durch ein Wort darstellbare Zahlbereich für die Anwendungen zu klein; man definiert dann binär-codierte ganze Zahlen durch eine Folge von mehr als zwei Bytes. Ein anderer Weg besteht darin, ganze Zahlen wie gewohnt zur Basis 10 darzustellen, was Konvertierungen bei Ein- und Ausgabe der Zahlen sehr einfach macht und besonders bei wenig rechenintensiven Vorgängen interessant ist. Diese beiden Wege wollen wir in den nächsten vier Unterkapiteln beschreiten.

Wer sich näher mit arithmetischen Algorithmen beschäftigen möchte, der findet viele interessante Überlegungen hierzu in dem Buch »Arithmetik in Rechenanlagen« von Otto Spaniol.

24.1 Binär-codierte vorzeichenlose ganze Zahlen

Binär-codierte vorzeichenlose ganze Zahlen werden durch Bitfolgen dargestellt, die in einem Stellenwertsystem mit Basis 2 zu interpretieren sind (siehe das Kapitel »Zahlsysteme«). Wegen der besseren Handhabung wollen wir nur Bitfolgen zulassen, die eine glatt durch 8 teilbare Anzahl von Bits enthalten; wir fassen die Bitfolgen dann als Bytefolgen auf und manipulieren sie entsprechend den Regeln für Felder. Wie bei Worten sind die niederwertigen Bytes in den niederwertigen Adressen abgelegt, die höherwertigen Bytes in den höherwertigen Adressen.

Die uns interessierenden Operationen sind die vier »Grundrechenarten« Addition, Subtraktion, Multiplikation und Division sowie für vorzeichenbehaftete ganze Zahlen das Komplementieren einer ganzen Zahl (unäres Minus). Wir werden dazu stets einige Zeiger auf die Bytefolgen benötigen, welche die ganzen Zahlen repräsentieren. Die Zeiger hält man günstigerweise in Registern. Da wir bis zu drei Zeiger brauchen, haben wir nur noch das A-Register für arithmetische und Zählvorgänge frei; dies ist entschieden zu wenig. Wir führen deshalb an die-

ser Stelle diejenigen Befehle des Z80 ein, mit denen der sekundäre Registersatz verfügbar gemacht wird. Prinzipiell sind die Algorithmen auch ohne den sekundären Registersatz formulierbar; sie werden allerdings wesentlich umständlicher.

Der Z80 verfügt über die sekundären Register A', F', B', C', D', E', H', L'. Diese Register können nicht direkt manipuliert werden; es gibt aber Befehle, welche die Hauptregister gegen die gleichnamigen sekundären Register austauschen:

Der Befehl

```
EX          AF,AF'
```

vertauscht das Registerpaar AF mit dem Registerpaar AF' (Doppelregister aus A' und F'). Der Befehl

```
EXX
```

vertauscht das Registerpaar BC gegen BC', DE gegen DE', HL gegen HL'. Die sekundären Register sind von den primären Registern nicht zu unterscheiden. Man muß also wissen, welcher Registersatz gerade aktiv ist. Das separate Vertauschen von AF und AF' ist sinnvoll, um Werte zwischen dem primären und dem sekundären Registersatz auszutauschen.

Wir treffen nun folgende Vereinbarung, die für den Rest des Kapitels gelten soll: Bei Operationen mit zwei Operanden zeigt das DE-Register auf den ersten Operanden, das HL-Register auf den zweiten Operanden, das BC-Register auf das Ergebnis; beim Komplementieren einer ganzen Zahl zeigt das HL-Register auf den Operanden, das BC-Register auf das Ergebnis.

Unser Akkumulator für arithmetische Operationen wird das A-Register. Alle weiteren Größen (Zähler etc.) bringen wir im sekundären Registersatz unter, den wir je nach Lage der Dinge mit dem primären Registersatz vertauschen.

Wir behandeln als erstes Addition und Subtraktion vorzeichenloser ganzer Zahlen, die eine feste Länge »L« besitzen; zur Darstellung des Ergebnisses steht ebenfalls wieder »L« Byte zur Verfügung. Bei der Durchführung der Addition kann es vorkommen, daß das Ergebnis nicht mit »L« Byte dargestellt werden kann; dies ist ein Fehler, der behandelt werden muß (Überlauf). Bei der Subtraktion ist es ein Fehler, wenn die zu subtrahierende Zahl größer ist als die Zahl, von der subtrahiert wird; das Ergebnis würde dadurch ja negativ und somit nicht darstellbar. In beiden Fällen werden wir in den Algorithmen nur einen Sprung auf eine Fehleradresse einbauen, uns aber mit der Fehlerbehandlung nicht weiter befassen.

Die Addition geht folgendermaßen vor sich: In einer Schleife über die gesamte Länge der Zahlen (»L« Durchlauf) bilden wir Byte-weise die Summe der beiden Operanden. Dabei kann ein Übertrag anfallen, der in den nächsten Schritt mit einbezogen werden muß. Um den ersten Schritt, bei dem kein Übertrag zu behandeln ist, genauso wie die restlichen Schritte ausführen zu können, starten wir mit einem fiktiven gelöschten Übertrag; der Übertrag wird stets im Übertrag-Flag aufbewahrt. Liegt nach Durchführung der »L«-ten Addition ein Übertrag vor, so signalisiert dieser einen Überlauf (Fehler).

Die feste Länge »L« denken wir uns als Konstante LAENGE (vom Typ »Byte«) vereinbart. Zu Beginn der Berechnung bringen wir »L« als Startwert für die Schleife in den Schleifenzähler B'.

```
        EXX                     ; sekundaeren Registersatz holen
        LD      B,LAENGE        ; Schleifenzaehler aufsetzen
        OR      A               ; Uebertrag-Flag loeschen
ADD:    EXX                     ; primaeren Registersatz holen
        LD      A,(DE)          ; Byte des ersten Operanden holen
        ADC     A,(HL)          ; Byte des zweiten Operanden
                                ; dazu addieren,
                                ; Uebertrag einbeziehen
        LD      (BC),A          ; Byte des Ergebnisses sichern
        INC     BC              ; auf jeweils
        INC     DE              ; naechstes Byte
        INC     HL              ; der Zahlen zeigen
        EXX                     ; sekundaeren Registersatz holen
        DJNZ    ADD             ; alle Bytes der Zahlen
                                ; bearbeiten
        EXX                     ; primaeren Registersatz holen
        JP      C,FEHLER        ; es ist Ueberlauf aufgetreten
```

Die Subtraktion läuft bis auf den eigentlichen arithmetischen Befehl exakt gleich ab:

```
        EXX                     ; sekundaeren Registersatz holen
        LD      B,LAENGE        ; Schleifenzaehler aufsetzen
        OR      A               ; Uebertrag-Flag loeschen
SUB:    EXX                     ; primaeren Registersatz holen
        LD      A,(DE)          ; Byte des ersten Operanden holen
        SBC     A,(HL)          ; Byte des zweiten Operanden
                                ; davon subtrahieren, eventuell
                                ; geborgtes Bit einbeziehen
        LD      (BC),A          ; Byte des Ergebnisses sichern
        INC     BC              ; auf jeweils
        INC     DE              ; naechstes Byte
        INC     HL              ; der Zahlen zeigen
        EXX                     ; sekundaeren Registersatz holen
        DJNZ    SUB             ; alle Bytes der Zahlen
                                ; bearbeiten
        EXX                     ; primaeren Registersatz holen
        JP      C,FEHLER        ; zu subtrahierende Zahl groesser
                                ; als Zahl, von der subtrahiert
                                ; werden soll
```

Die Multiplikation zweier vorzeichenloser ganzer Zahlen führen wir durch mehrfaches Addieren und Verschieben des Zwischenergebnisses durch; ein Beispiel für diese Technik haben wir bereits im Kapitel 13.1 betrachtet.

Multiplikator und Multiplikand sollen jeweils die Länge »L« (in Bytes) haben. Das Ergebnis der Multiplikation belegt damit höchstens 2 * »L« Bytes; wir nehmen deshalb an, daß für das Ergebnis 2 * »L« Bytes bereitgestellt wurden. Soll das Ergebnis wieder mit »L« Byte dargestellt werden, so tritt möglicherweise ein Überlauf auf; diese Situation lassen wir hier jedoch außer acht. Die Größe »L« stellen wir wieder durch die Konstante LAENGE dar.

Vor Beginn der Addition muß der Akkumulator gelöscht werden; dies leistet folgendes Unterprogramm, das im HL-Register einen Zeiger auf das niederwertigste Byte des Akkumulators erhält:

```
LOESCH:  PUSH     HL          ; Registerinhalte
         PUSH     BC          ; sichern
         LD       B,2*LAENGE  ; Laenge des Akkumulators laden
NULL:    LD       (HL),0      ; Byte des Akkumulators loeschen
         INC      HL          ; auf naechstes Byte des
                              ; Akkumulators zeigen
         DJNZ     NULL        ; gesamten Akkumulator loeschen
         POP      BC          ; Register
         POP      HL          ; restaurieren
         RET
```

Für das Aufaddieren des Multiplikanden auf das Zwischenergebnis verwenden wir folgendes Unterprogramm, das im HL-Register einen Zeiger auf das niederwertigste Byte des Zwischenergebnisses, im DE-Register einen Zeiger auf das niederwertigste Byte des Multiplikanden erhält:

```
ADD:     PUSH     BC          ; Register-
         PUSH     DE          ; inhalte
         PUSH     HL          ; sichern
         OR       A           ; Uebertrag-Flag loeschen
         LD       B,LAENGE    ; Laenge des Multiplikanden laden
ADDB:    LD       A,(DE)      ; Byte des Multiplikanden holen
         ADC      A,(HL)      ; Byte des Zwischenergebnisses
                              ; hinzuaddieren
         LD       (HL),A      ; Byte ins Zwischenergebnis
                              ; zurueckschreiben
         C        DE          ; auf naechstes Byte des
                              ; Multiplikanden zeigen
         INC      HL          ; auf naechstes Byte des
                              ; Zwischenergebnisses zeigen
         DJNZ     ADDB        ; alle Bytes des Multiplikanden
                              ; zum Zwischenergebnis addieren
         LD       B,LAENGE    ; restliche Laenge des
                              ; Zwischenergebnisses laden
```

```
            JP      NC,FERTIG   ; kein Uebertrag mehr zu
                                ; beruecksichtigen
UEBERT:     INC     (HL)        ; Uebertrag von vorhergehender
                                ; Stelle beruecksichtigen
            JP      NZ,FERTIG   ; kein weiterer Uebertrag
                                ; zu beruecksichtigen
            INC     HL          ; auf naechstes Byte des
                                ; Zwischenergebnisses zeigen
            DJNZ    UEBERT      ; Uebertrag eventuell durch alle
                                ; Bytes des Zwischenergebnisses
                                ; ziehen
FERTIG:     POP     HL          ; alle
            POP     DE          ; Register
            POP     BC          ; restaurieren
            RET
```

Eine (arithmetische) Linksverschiebung des Zwischenergebnisses um ein Bit realisieren wir durch folgende Routine, welche wieder im HL-Register einen Zeiger auf das niederwertigste Byte des Zwischenergebnisses erhält:

```
SCHIEB:     PUSH    HL          ; Registerinhalte
            PUSH    BC          ; sichern
            LD      B,2*LAENGE  ; Laenge des Zwischenergebnisses
                                ; laden
            OR      A           ; Uebertrag-Flag loeschen
SBYTE:      RL      (HL)        ; Byte um ein Bit
                                ; linksverschieben, alten Wert
                                ; des Uebertrag-Flags einfuegen,
                                ; hoechstes Bit ins
                                ; Uebertrag-Flag bringen
            INC     HL          ; auf naechstes Byte des
                                ; Zwischenergebnisses zeigen
            DJNZ    SBYTE       ; alle Bytes des
                                ; Zwischenergebnisses bearbeiten
            POP     BC          ; Register
            POP     HL          ; wiederherstellen
            RET
```

Vor Beginn der eigentlichen Multiplikation bringen wir erst einige Werte in sekundäre Register; aus dem Zeiger auf das niederwertigste Byte des Multiplikators machen wir einen Zeiger direkt hinter das höchstwertigste Byte des Multiplikators:

```
            PUSH    DE          ; Zeiger auf Multiplikand und
            PUSH    BC          ; Zeiger auf Ergebnis auf den
```

```
                                    ; Stapel bringen zwecks
                                    ; Einbringung in sekundaeren
                                    ; Registersatz
        LD      DE,LAENGE   ; Zeiger auf Byte hinter dem
        ADD     HL,DE       ; hoechstwertigen Byte des
                            ; Multiplikators berechnen
        EXX                 ; sekundaeren Registersatz holen
        POP     HL          ; Zeiger auf Ergebnis und
        POP     DE          ; Zeiger auf Multiplikand holen
        CALL    LOESCH      ; Akkumulator loeschen
        EXX                 ; primaeren Registersatz holen
        LD      B,LAENGE    ; Anzahl der Bytes des
                            ; Multiplikators laden
MULT:   DEC     HL          ; auf naechstes Byte des
                            ; Multiplikators zeigen
        EXX                 ; sekundaeren Registersatz holen
        LD      B,8         ; Anzahl der Bits pro Byte laden
BMULT:  CALL    SCHIEB      ; Zwischenergebnis um ein Bit
                            ; linksschieben
        EXX                 ; primaeren Registersatz holen
        RLC     (HL)        ; Bit des Multiplikators holen
        EXX                 ; sekundaeren Registersatz holen
        CALL    C,ADD       ; Bit war gesetzt, Multiplikand
                            ; zum Zwischenergebnis addieren
        DJNZ    BMULT       ; alle Bits dieses Bytes des
                            ; Multiplikators verarbeiten
        EXX                 ; primaeren Registersatz holen
        DJNZ    MULT        ; alle Bytes des Multiplikators
                            ; verarbeiten
```

Der Multiplikationsprozeß kann beschleunigt werden. Wir betrachten folgendes Phänomen, das uns auf das Verfahren von *Booth* führt: wenn eine Serie von Null-Bits im Multiplikator auftritt, so erfolgt eine entsprechende Anzahl von Linksverschiebungen ohne Addition. Analog dazu betrachten wir eine Folge von Eins-Bits, zum Beispiel die Bits $b_j, b_{j-1}, ..., b_{i+1}, b_i$. Für jedes dieser Eins-Bits müßte eine Addition des Multiplikanden zum Zwischenergebnis und ein anschließendes Linksverschieben des Zwischenergebnisses um ein Bit erfolgen. Dies ist aber äquivalent zu der Methode, an der Stelle i den Multiplikanden vom Zwischenergebnis zu subtrahieren, an den Stellen i+1 bis j weder zu addieren noch zu subtrahieren, und an der Stelle j+1 den Multiplikanden zum Zwischenergebnis zu addieren. Da wir den Multiplikator von vorne abtasten, entspricht ein Übergang von Null nach Eins einer Addition, ein Übergang von Eins nach Null einer Subtraktion. Den Multiplikator denken wir uns nach vorne und hinten um ein Null-Bit verlängert.

Wir stellen den Algorithmus in einem Flußdiagramm dar:

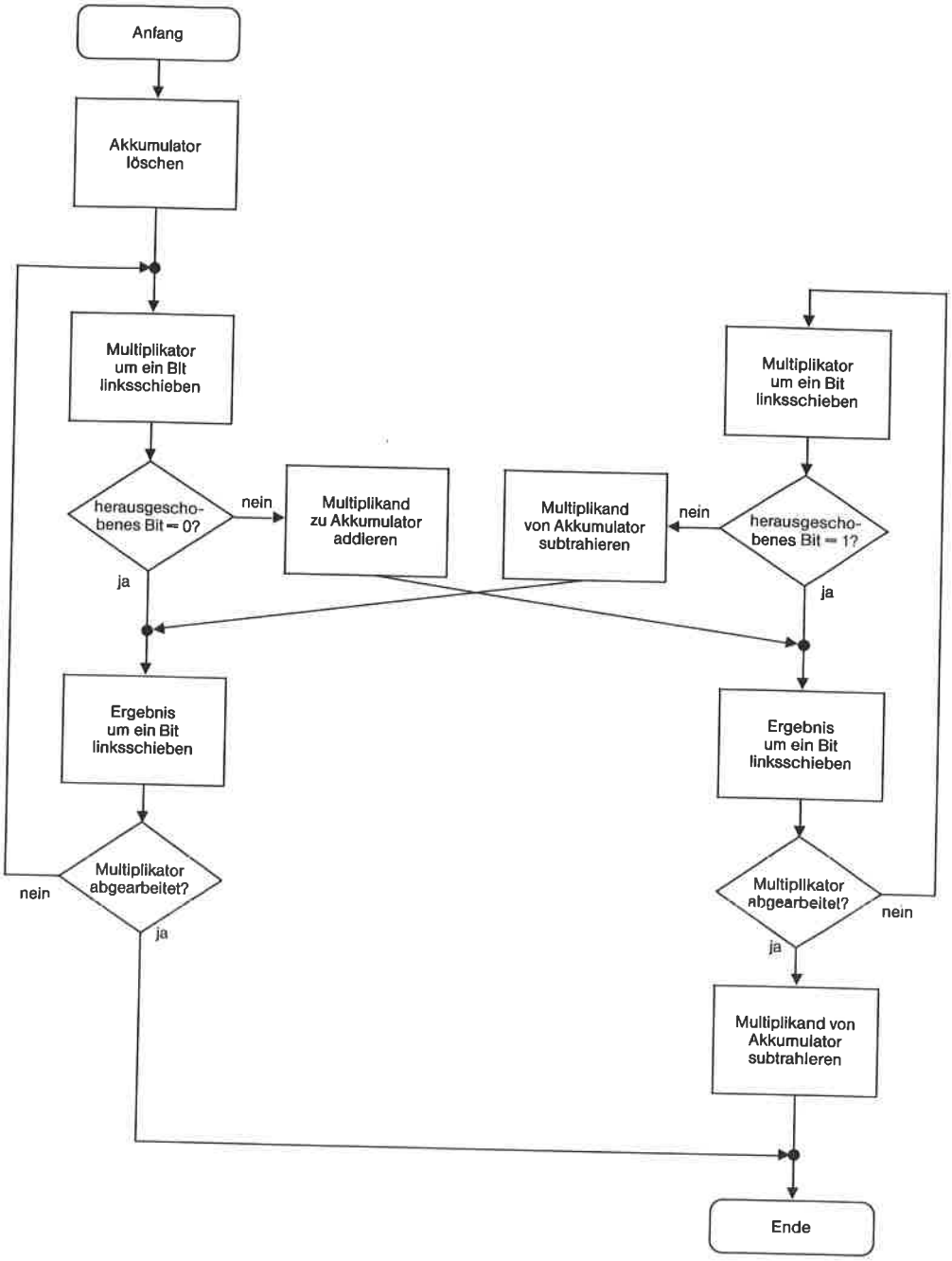


Bild 24.1. *Flußdiagramm: Schnelle Multiplikation nach Booth*

Die Effizienz des Algorithmus wird aus folgendem Beispiel klar: Wenn der Multiplikator den Wert 1111100001111111B besitzt, so sind nach dem normalen Multiplikationsalgorithmus 12 Additionen auszuführen, nach dem Verfahren von Booth dagegen nur 2 Additionen und 2 Subtraktionen. Die Subtraktion erfolgt ähnlich wie die Addition:

```
SUB:      PUSH    BC          ; Register-
          PUSH    DE          ; inhalte
          PUSH    HL          ; sichern
          OR      A           ; Uebertrag-Flag loeschen
          LD      B,LAENGE    ; Laenge des Multiplikanden laden
          EX      DE,HL       ; Zeiger tauschen
SUBB:     LD      A,(DE)      ; Byte des Zwischenergebnisses
                              ; holen
          SBC     A,(HL)      ; Byte des Multiplikanden
                              ; abziehen
          LD      (DE),A      ; Byte ins Zwischenergebnis
                              ; zurueckschreiben
          INC     HL          ; auf naechstes Byte des
                              ; Multiplikanden zeigen
          NC      DE          ; auf naechstes Byte des
                              ; Zwischenergebnisses zeigen
          DJNZ    SUBB        ; alle Bytes des Multiplikanden
                              ; vom Zwischenergebnis
                              ; subtrahieren
          LD      B,LAENGE    ; restliche Laenge des
                              ; Zwischenergebnisses laden
          EX      DE,HL       ; Zeiger tauschen
BORGEN:   JP      NC,FERTIG   ; kein Borgen mehr zu
                              ; beruecksichtigen
          LD      A,(HL)      ; Borgen von vorhergehender
          SBC     A,0         ; Stelle
          LD      (HL),A      ; beruecksichtigen
          INC     HL          ; auf naechstes Byte des
                              ; Zwischenergebnisses zeigen
          DJNZ    BORGEN      ; Borgen eventuell durch alle
                              ; Bytes des Zwischenergebnisses
                              ; ziehen
FERTIG:   POP     HL          ; alle
          POP     DE          ; Register
          POP     BC          ; restaurieren
          RET
```

Das Verfahren von Booth lautet damit:

```
        PUSH    DE          ; Zeiger auf Multiplikand und
        PUSH    BC          ; Zeiger auf Ergebnis auf den
                            ; Stapel bringen zwecks
                            ; Einbringung in sekundaeren
                            ; Registersatz
        LD      DE,LAENGE   ; Zeiger auf Byte hinter dem
        ADD     HL,DE       ; hoechstwertigen Byte des
                            ; Multiplikators berechnen
        EXX                 ; sekundaeren Registersatz holen
        POP     HL          ; Zeiger auf Ergebnis und
        POP     DE          ; Zeiger auf Multiplikand holen
        CALL    LOESCH      ; Akkumulator loeschen
        EXX                 ; primaeren Registersatz holen
        LD      B,LAENGE    ; Anzahl der Bytes des
                            ; Multiplikators laden
LMULT:  DEC     HL          ; auf naechstes Byte des
                            ; Multiplikators zeigen
        EXX                 ; sekundaeren Registersatz holen
        LD      B,8         ; Anzahl der Bits pro Byte laden
LBMULT: EXX                 ; primaeren Registersatz holen
        RLC     (HL)        ; Bit des Multiplikators holen
        EXX                 ; sekundaeren Registersatz holen
        JP      C,ADDIER    ; Ende einer Folge von Nullen,
                            ; Addition ausführen, in rechten
                            ; Teil des Programms wechseln
LTEST:  CALL    SCHIEB      ; Zwischenergebnis um ein Bit
                            ; linksschieben
        DJNZ    LBMULT      ; alle Bits dieses Bytes des
                            ; Multiplikators verarbeiten
        EXX                 ; primaeren Registersatz holen
        DJNZ    LMULT       ; alle Bytes des Multiplikators
                            ; verarbeiten
        JP      FERTIG      ; Multiplikation ausgeführt
ADDIER: CALL    ADD         ; Multiplikand zu
                            ; Zwischenergebnis addieren
        JP      RTEST       ; in rechten Teil des Programms
                            ; wechseln
SUBTRA: CALL    SUB         ; Multiplikand vom
                            ; Zwischenergebnis subtrahieren
        JP      LTEST       ; in linken Teil des Programms
                            ; wechseln
RMULT:  DEC     HL          ; auf naechstes Byte des
                            ; Multiplikators zeigen
```

```
        EXX                     ; sekundaeren Registersatz holen
        LD      B,8             ; Anzahl der Bits pro Byte laden
RBMULT: EXX                     ; primaeren Registersatz holen
        RLC     (HL)            ; Bit des Multiplikators holen
        EXX                     ; sekundaeren Registersatz holen
        JP      NC,SUBTRA       ; Ende einer Folge von Einsen,
                                ; Subtraktion ausführen,
                                ; in linken Teil
                                ; des Programms wechseln
RTEST:  CALL    SCHIEB          ; Zwischenergebnis um ein Bit
                                ; linksschieben
        DJNZ    RBMULT          ; alle Bits dieses Bytes des
                                ; Multiplikators verarbeiten
        EXX                     ; primaeren Registersatz holen
        DJNZ    RMULT           ; alle Bytes des Multiplikators
                                ; verarbeiten
        EXX                     ; sekundaeren Registersatz holen
        CALL    SUB             ; Multiplikand vom
                                ; Zwischenergebnis subtrahieren
        EXX                     ; primaeren Registersatz holen
FERTIG: NOP                     ; gemeinsame Fortsetzungsstelle
```

Das Verfahren von Booth kann nochmals beschleunigt werden, wenn man die Fälle, in denen eine isolierte Eins oder Null in der Bitfolge des Multiplikators auftritt, gesondert behandelt (Näheres siehe: Otto Spaniol).

Die Multiplikation mit einer Zweierpotenz läßt sich noch schneller durch Linksverschiebungen bewerkstelligen; für den Multiplikator 2^r müssen r Linksverschiebungen des Multiplikanden getätigt werden. Einen Überlauf erkennt man bei dieser Methode am Übertrag-Flag.

Bei der Division zweier ganzer Zahlen der Länge »L« (Bytes) fällt ein Quotient der Länge »L« an. Ist der Dividend kein ganzzahliges Vielfaches des Divisors, so ergibt sich ein Divisionsrest; dieser soll ebenfalls stets die Länge »L« besitzen.

Es gibt relativ viele Möglichkeiten, die Division auszuführen (für eine ausführliche Darstellung sei auf das Buch von Otto Spaniol verwiesen); wir zeigen eine einfache Methode, die mit bedingten Subtraktionen arbeitet.

Der Quotient erhält zunächst den Wert Null.

Nun müssen wir den Divisor normieren, das heißt ihn so lange nach links verschieben, bis das höchstwertige Bit den Wert 1 trägt; dabei wird t-mal das höchstwertige Bit des Divisors getestet. Wenn nach 8»L« Tests und Linksverschiebungen das höchstwertige Bit des Divisors immer noch den Wert 0 besitzt, so hat der Divisor den Wert Null; die »Division durch Null« wird dann mit Fehler abgebrochen.

Als nächstes folgen t Divisionsschritte. In jedem Schritt wird zunächst der Quotient um ein Bit linksverschoben. Dann ziehen wir den (normierten) Divisor vom Rest des Dividenden ab. Geht das ohne Überlauf vonstatten, so erhält das niederwertigste des Quotienten den Wert 1.

Trat ein Überlauf bei der Subtraktion auf, so addieren wir den Divisor wieder zum Divisionsrest; das niederwertigste Bit des Quotienten erhält den Wert 0. Abschließend wird der Rest des Dividenden um ein Bit linksverschoben; es kann dabei vorkommen, daß bei der letzten Verschiebung der Wert 1 ins Übertrag-Flag gelangt. Das Übertrag-Flag stellt damit das höchstwertige Bit des Dividenden dar und muß bei der Subtraktion beziehungsweise der Addition berücksichtigt werden.

Zuletzt stellen wir den Rest der Division durch t-1 logische Rechtsverschiebungen des Dividenden stellenwertkorrekt dar.

Zu Beginn zeigen DE, HL, BC auf den Dividenden, Divisor, Quotienten. Nach Abschluß der Division zeigt DE auf den Divisionsrest, HL auf den normierten Divisor und BC auf den Quotienten.

```
        PUSH    HL              ; Zeiger auf Divisor sichern
        PUSH    BC              ; Zeiger auf
        EXX                     ; Quotienten in
        POP     HL              ; sekundaeres Register bringen
        LD      B,LAENGE        ; Laenge des Quotienten laden
QULOES: LD      (HL),0          ; Byte des Quotienten loeschen
        INC     HL              ; auf naechstes Byte
                                ; des Quotienten zeigen
        DJNZ    QULOES          ; gesamten Quotienten loeschen
        POP     DE              ; Zeiger auf Divisor holen
        LD      HL,LAENGE-1     ; Zeiger auf MSB
        ADD     HL,DE           ; des Divisors berechnen
        XOR     A               ; Anzahl der Test ruecksetzen
NORM:   INC     A               ; Anzahl der Tests um 1 erhoehen
        BIT     7,(HL)          ; hoechstes Bit des Divisors
                                ; testen
        JP      NZ,NORMIE       ; Divisor ist normiert
        EX      DE,HL           ; Zeiger auf Divisor holen
        OR      A               ; Uebertrag-Flag loeschen
        LD      B,LAENGE        ; Laenge des Divisors laden
        PUSH    HL              ; Zeiger auf Divisor sichern
SCHIDR: RL      (HL)            ; Byte des Divisors
                                ; linksverschieben
        INC     HL              ; auf naechstes Byte
                                ; des Divisors zeigen
        DJNZ    SCHIDR          ; gesamten Divisor
                                ; linksverschieben
        POP     HL              ; Zeiger auf Divisor restaurieren
        EX      DE,HL           ; und sichern
        CP      8*LAENGE        ; Anzahl der Tests pruefen
        JP      NZ,NORM         ; Normierung fortsetzen
```

```
            JP      FEHLER      ; Fehler: Division durch Null
NORMIE:     EXX                 ; primaeren Registersatz holen
            PUSH    BC          ; Zeiger auf Quotienten sichern
            LD      C,A         ; Anzahl der Divisionsschritte
                                ; kopieren
            OR      A           ; hoechstes Bit des Dividenden
                                ; loeschen
            EX      AF,AF'      ; Anzahl der Divisionsschritte
                                ; und hoechstes Bit des
                                ; Dividenden sichern
QUSCHI:     OR      A           ; Uebertrag-Flag loeschen
            LD      B,LAENGE    ; Laenge des Quotienten
            EX      (SP),HL     ; Zeiger auf Quotienten holen,
                                ; Zeiger auf Divisor sichern
            PUSH    HL          ; Zeiger auf Quotienten sichern
SCHIQU:     RL      (HL)        ; Byte des Quotienten
                                ; linksverschieben
            INC     HL          ; auf naechstes Byte
                                ; des Quotienten zeigen
            DJNZ    SCHIQU      ; gesamten Quotienten
                                ; linksverschieben
            POP     HL          ; Zeiger auf Quotienten holen
            EX      (SP),HL     ; Zeiger auf Divisor holen,
                                ; Zeiger auf Quotienten sichern
            OR      A           ; Uebertrag-Flag loeschen
            LD      B,LAENGE    ; Laenge des Dividenden
            PUSH    HL          ; Zeiger auf Divisor und
            PUSH    DE          ; Zeiger auf Dividenden sichern
SUBTRA:     LD      A,(DE)      ; Byte des
            SBC     A,(HL)      ; Divisors von Byte
            LD      (DE),A      ; des Dividenden subtrahieren
            INC     DE          ; auf naechstes Byte von
            INC     HL          ; Divisor und Dividend zeigen
            DJNZ    SUBTRA      ; Divisor von Dividend
                                ; subtrahieren
            POP     DE          ; Zeiger auf Dividend und
            POP     HL          ; Divisor restaurieren
            JP      NC,QUOT1    ; kein Ueberlauf, Subtraktion
                                ; erfolgreich
            EX      AF,AF'      ; hoechstes Bit des Dividenden
                                ; holen
            JP      NC,QUOT0    ; Ueberlauf bei Subtraktion,
                                ; rueckgaengig machen
```

```
                CCF                     ; hoechstes Bit des Dividenden
                                        ; loeschen
                EX      AF,AF'          ; und sichern
QUOT1:          EX      (SP),HL         ; Zeiger auf Quotienten holen,
                                        ; Zeiger auf Dividenden sichern
                INC     (HL)            ; niederwertiges Bit des
                                        ; Quotienten setzen
                EX      (SP),HL         ; Zeiger auf Dividenden holen,
                                        ; Zeiger auf Quotienten sichern
                JP      DDSCHI          ; Dividenden linksverschieben
QUOTO:          EX      AF,AF'          ; hoechstes Bit des Dividenden
                                        ; sichern
                OR      A               ; Uebertrag-Flag loeschen
                LD      B,LAENGE        ; Laenge des Dividenden laden
                PUSH    HL              ; Zeiger auf Divisor und
                PUSH    DE              ; Zeiger auf Dividend sichern
ADDIER:         LD      A,(DE)          ; Byte des
                ADC     A,(HL)          ; Divisors zu Byte
                LD      (DE),A          ; des Dividenden addieren
                INC     DE              ; auf naechstes Byte von
                INC     HL              ; Divisor und Dividend zeigen
                DJNZ    ADDIER          ; Divisor zu Dividend addieren
                POP     DE              ; Zeiger auf Dividend und
                POP     HL              ; Divisor restaurieren
DDSCHI:         EX      AF,AF'          ; hoechstes Bit des Dividenden
                                        ; holen, dieses hat Wert Null
                LD      B,LAENGE        ; Laenge des Dividenden laden
                EX      DE,HL           ; Zeiger auf Dividend holen
                PUSH    HL              ; und sichern
SCHIDD:         RL      (HL)            ; Byte des Dividenden
                                        ; linksverschieben
                INC     HL              ; auf naechstes Byte
                                        ; des Dividenden zeigen
                DJNZ    SCHIDD          ; gesamten Dividenden
                                        ; linksverschieben
                EX      AF,AF'          ; hoechstes Bit des Dividenden
                                        ; sichern
                POP     HL              ; Zeiger auf Dividend
                                        ; restaurieren
                EX      DE,HL           ; Zeiger auf Divisor holen
                DEC     C               ; Anzahl der Divisionsschritte
                                        ; um eins vermindern
                JP      NZ,QUSCHI       ; weiteren Divisionsschritt
```

```
                                        ; ausfuehren
        EX      DE,HL       ; Zeiger auf Rest holen
        PUSH    HL          ; und sichern
        LD      BC,LAENGE-1 ; Zeiger auf MSB des Rests
        ADD     HL,BC       ; berechnen
        EX      AF,AF'      ; Anzahl der Rechtsverschiebungen
                            ; des Rests + 1 holen
        JP      TEST        ; in abweisende Schleife springen
RENORM: LD      B,LAENGE    ; Laenge des Rests
        PUSH    HL          ; Zeiger auf MSB des Rests
                            ; sichern
RESCHI: RR      (HL)        ; Byte des Rests
                            ; rechtsverschieben
        DEC     HL          ; auf naechstes Byte
                            ; des Rests zeigen
        DJNZ    RESCHI      ; gesamten Rest rechtsverschieben
        POP     HL          ; Zeiger auf MSB des Rests holen
TEST:   DEC     A           ; Anzahl der restlichen
                            ; Rechtsverschiebungen des
                            ; Divisionsrests berechnen
        JP      NZ,RENORM   ; weitere Rechtsverschiebung
                            ; ausfuehren
        POP     HL          ; Zeiger auf Rest restaurieren
        EX      DE,HL       ; Zeiger tauschen
        POP     BC          ; Zeiger auf Quotienten
                            ; restaurieren
```

Die Division durch eine Zweierpotenz erledigen wir durch Rechtsverschiebungen. Hat der Divisor den Wert 2^r, so sind r logische Rechtsverschiebungen des Dividenden nötig, um den Quotienten zu berechnen; der Rest der Division besteht aus den r Bits, die aus dem Dividenden hinausgeschoben wurden.

Übungen

1. In den Multiplikationsalgorithmen wurde stets der gesamte Akkumulator bearbeitet, obwohl das Zwischenergebnis erst gegen Ende des Verfahrens seine volle Länge erreicht. Optimiere die Algorithmen so, daß nur der wirklich belegte Teil des Akkumulators bearbeitet wird.

24.2 Binär-codierte vorzeichenbehaftete ganze Zahlen

Unter binär-codierten vorzeichenbehafteten ganzen Zahlen verstehen wir ganze Zahlen in 2-Komplement-Darstellung (siehe Kapitel 2).

Addition und Subtraktion binär-codierter vorzeichenbehafteter ganzer Zahlen verlaufen algorithmisch fast genau wie Addition und Subtraktion binär-codierter vorzeichenloser Zahlen. Einen Überlauf erkennen wir allerdings hier nicht am Übertrag-Flag, sondern am Überlauf-Flag. Ein Überlauf tritt auf, wenn das Ergebnis der Operation nicht mit der vorgegebenen Länge von »L« Byte dargestellt werden kann; dies bedeutet eine Überschreitung des Zahlbereichs in positiver oder in negativer Richtung. Bei der Subtraktion können wir am Überlauf nicht feststellen, welcher der beiden Operanden der größere war; wir müssen dazu das Vorzeichen-Flag mit einbeziehen: <P> xor <S> = 1 genau dann, wenn der zweite Operand größer war.

Die Routine für die Addition lautet:

```
        EXX
        LD      B,LAENGE    ; sekundaeren Registersatz holen
        OR      A           ; Schleifenzaehler aufsetzen
ADD:    EXX                 ; Uebertrag-Flag loeschen
                            ; primaeren Registersatz holen
        ADC     A,(DE)      ; Byte des ersten Operanden holen
                A,(HL)      ; Byte des zweiten Operanden
                            ; dazu addieren,
                            ; Uebertrag einbeziehen
        LD      (BC),A      ; Byte des Ergebnisses sichern
        INC     BC          ; auf jeweils
        INC     DE          ; naechstes Byte
        INC     HL          ; der Zahlen zeigen
        EXX                 ; sekundaeren Registersatz holen
        DJNZ    ADD         ; alle Bytes der Zahlen
                            ; bearbeiten
        EXX                 ; primaeren Registersatz holen
        JP      PE,FEHLER   ; es ist Ueberlauf aufgetreten
```

Die Routine für die Subtraktion lautet:

```
        EXX
        LD      B,LAENGE    ; sekundaeren Registersatz holen
        OR      A           ; Schleifenzaehler aufsetzen
SUB:    EXX                 ; Uebertrag-Flag loeschen
                            ; primaeren Registersatz holen
        LD      A,(DE)      ; Byte des ersten Operanden holen
        SBC     A,(HL)      ; Byte des zweiten Operanden
                            ; davon subtrahieren, eventuell
                            ; geborgtes Bit einbeziehen
        LD      (BC),A      ; Byte des Ergebnisses sichern
```

```
        INC     BC          ; auf jeweils
        INC     DE          ; naechstes Byte
        INC     HL          ; der Zahlen zeigen
        EXX                 ; sekundaeren Registersatz holen
        DJNZ    SUB         ; alle Bytes der Zahlen
                            ; bearbeiten
        EXX                 ; primaeren Registersatz holen
        JP      PE,FEHLER   ; es ist Ueberlauf aufgetreten
```

Um eine Zahl zu komplementieren, können wir einen Algorithmus verwenden, der wie die Subtraktion arbeitet, als ersten Operanden aber eine Null einsetzt:

```
        EXX                 ; sekundaeren Registersatz holen
        LD      B,LAENGE    ; Schleifenzaehler aufsetzen
        OR      A           ; Uebertrag-Flag loeschen
KOMPL:  EXX                 ; primaeren Registersatz holen
        LD      A,0         ; ersten Operanden Null setzen
        SBC     A,(HL)      ; Byte des zweiten Operanden
                            ; davon subtrahieren, eventuell
                            ; geborgtes Bit einbeziehen
        LD      (BC),A      ; Byte des Ergebnisses sichern
        INC     BC          ; auf jeweils naechstes
        INC     HL          ; Byte der Zahlen zeigen
        EXX                 ; sekundaeren Registersatz holen
        DJNZ    KOMPL       ; alle Bytes der Zahlen
                            ; bearbeiten
        EXX                 ; primaeren Registersatz holen
        JP      PE,FEHLER   ; es ist Ueberlauf aufgetreten
```

Da der Zahlbereich unsymmetrisch ist, tritt ein Überlauf auf, wenn die kleinste darstellbare Zahl komplementiert werden soll.

Bei der Multiplikation wirkt sich das 2-Komplement recht hinderlich aus. Wir sehen uns zunächst an, was sich bei formaler Multiplikation zweier Zahlen im 2-Komplement - interpretiert als vorzeichenlose ganze Zahlen - ergibt. Eine nichtnegative Zahl t wird im 2-Komplement mit n Bits durch t dargestellt, eine negative Zahl t durch $2^n + t$. Das Ergebnis z (modulo 2^{2n}) der formalen Multiplikation besitzt $2 * n$ Bits. Wir erhalten bei formaler Multiplikation vier Fälle:

1. $x >= 0, y >= 0$: $z = x * y$
2. $x >= 0, y < 0$: $z = x * y + 2^n * x$
3. $x < 0, y >= 0$: $z = x * y + 2^n * y$
4. $x < 0, y < 0$: $z = (x * y + 2^n * (x + y) + 2^{2n}) \bmod 2^{2n}$
   $= x * y + 2^n * (x + y)$

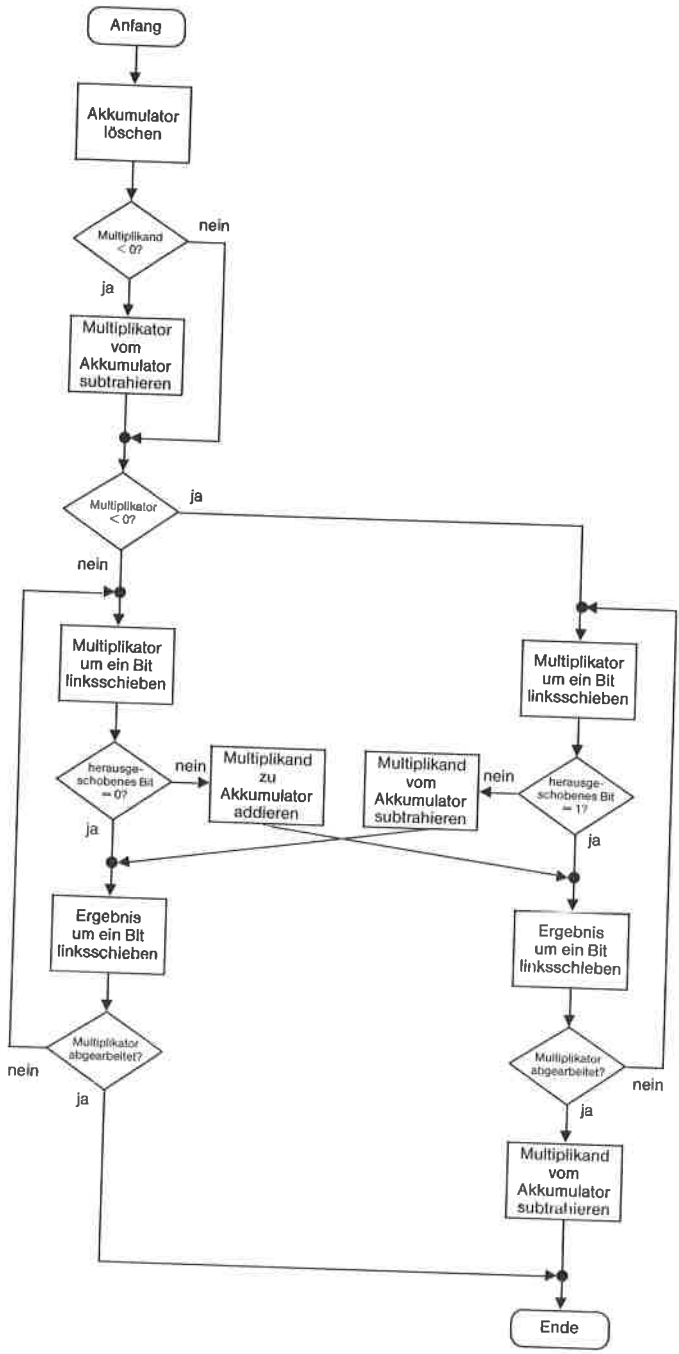


Bild 24.2. *Flußdiagramm: Schnelle Multiplikation vorzeichenbehafteter ganzer Zahlen*

Wir ersehen daraus, daß wir das 2^nfache des Multiplikators vom formalen Produkt abziehen müssen, falls der Multiplikand negativ ist, das 2^nfache des Multiplikanden, wenn der Multiplikator negativ ist.

Wir prüfen als erstes den Multiplikanden. Ist dieser negativ, so subtrahieren wir den Multiplikator vom Akkumulator; durch die später folgenden Linksverschiebungen wird diese Operation stellenwertkorrekt ausgeführt.

Ist der Multiplikator negativ, so steigen wir anstelle der nötigen Subtraktion des Multiplikanden – abweichend vom Multiplikationsalgorithmus aus dem vorhergehenden Unterkapitel – in den rechten Teil des Verfahrens von Booth ein.

Wir erhalten damit umstehendes Flußdiagramm.

Unter Verwendung der Unterprogramme aus dem vorhergehenden Unterkapitel lautet das Multiplikationsprogramm:

```
PUSH    DE          ; Zeiger auf Multiplikand,
PUSH    HL          ; Zeiger auf Multiplikator,
PUSH    DE          ; Zeiger auf Multiplikand und
PUSH    BC          ; Zeiger auf Ergebnis auf den
                    ; Stapel bringen zwecks
                    ; Einbringung in sekundaeren
                    ; Registersatz
LD      BC,LAENGE-1 ; Zeiger auf
ADD     HL,BC       ; hoechstwertiges Byte des
                    ; Multiplikators berechnen
EXX                 ; sekundaeren Registersatz holen
POP     HL          ; Zeiger auf Ergebnis und
POP     DE          ; Zeiger auf Multiplikand holen
CALL    LOESCH      ; Akkumulator loeschen
EX      DE,HL       ; Zeiger auf Ergebnis sichern
LD      BC,LAENGE-1 ; Zeiger auf hoechstwertiges Byte
ADD     HL,BC       ; des Multiplikanden berechnen
BIT     7,(HL)      ; Vorzeichen des Multiplikanden
                    ; testen
EX      DE,HL       ; Zeiger auf Ergebnis
                    ; restaurieren
POP     DE          ; Zeiger auf Multiplikator holen
CALL    NZ,SUB      ; Korrektur fuer negativen
                    ; Multiplikanden
POP     DE          ; Zeiger auf Multiplikand holen
EXX                 ; primaeren Registersatz holen
LD      B,LAENGE    ; Anzahl der Bytes des
                    ; Multiplikators laden
```

```
        BIT     7,(HL)      ; Vorzeichen des Multiplikators
                            ; testen
        INC     HL          ; auf Byte hinter Multiplikator
                            ; zeigen
        JP      NZ,RMULT    ; Multiplikator negativ,
                            ; in rechten Teil des
                            ; Algorithmus einsteigen
LMULT:  DEC     HL          ; auf naechstes Byte des
                            ; Multiplikators zeigen
        EXX                 ; sekundaeren Registersatz holen
        LD      B,8         ; Anzahl der Bits pro Byte laden
LBMULT: EXX                 ; primaeren Registersatz holen
        RLC     (HL)        ; Bit des Multiplikators holen
        EXX                 ; sekundaeren Registersatz holen
        JP      C,ADDIER    ; Ende einer Folge von Nullen,
                            ; Addition ausführen, in rechten
LTEST:  CALL    SCHIEB      ; Teil des Programms wechseln
                            ; Zwischenergebnis um ein Bit
        DJNZ    LBMULT      ; linksschieben
                            ; alle Bits dieses Bytes des
        EXX                 ; Multiplikators verarbeiten
        DJNZ    LMULT       ; primaeren Registersatz holen
                            ; alle Bytes des Multiplikators
        JP      FERTIG      ; verarbeiten
ADDIER: CALL    ADD         ; Multiplikation ausgeführt
                            ; Multiplikand zu
        JP      RTEST       ; Zwischenergebnis addieren
                            ; in rechten Teil des Programms
SUBTRA: CALL    SUB         ; wechseln
                            ; Multiplikand vom
        JP      LTEST       ; Zwischenergebnis subtrahieren
                            ; in linken Teil des Programms
RMULT:  DEC     HL          ; wechseln
                            ; auf naechstes Byte des
        EXX                 ; Multiplikators zeigen
        LD      B,8         ; sekundaeren Registersatz holen
RBMULT: EXX                 ; Anzahl der Bits pro Byte laden
        RLC     (HL)        ; primaeren Registersatz holen
        EXX                 ; Bit des Multiplikators holen
        JP      NC,SUBTRA   ; sekundaeren Registersatz holen
                            ; Ende einer Folge von Einsen,
                            ; Subtraktion ausführen,
                            ; in linken Teil
```

```
                                    ; des Programms wechseln
RTEST:  CALL    SCHIEB      ; Zwischenergebnis um ein Bit
                            ; linksschieben
        DJNZ    RBMULT      ; alle Bits dieses Bytes des
                            ; Multiplikators verarbeiten
        EXX                 ; primaeren Registersatz holen
        DJNZ    RMULT       ; alle Bytes des Multiplikators
                            ; verarbeiten
        EXX                 ; sekundaeren Registersatz holen
        CALL    SUB         ; Multiplikand von
                            ; Zwischenergebnis subtrahieren
        EXX                 ; primaeren Registersatz holen
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Die Multiplikation mit einer Zweierpotenz 2^r kann wieder durch r Linksverschiebungen bewerkstelligt werden; einen Überlauf erkennt man daran, daß nach der letzten Linksverschiebung Vorzeichen-Flag und Übertrag-Flag verschiedene Werte haben.

Für die Division vorzeichenbehafteter ganzer Zahlen empfiehlt sich die Umrechnung negativer Zahlen in die Vorzeichen/Betrag-Darstellung mit getrennter Behandlung von Vorzeichen und Betrag; dies läßt sich zum Beispiel mit obenstehender Komplementierungsroutine durchführen. Die Division der Beträge führen wir wie in Unterkapitel 24.1 durch; anschließend muß eventuell ins 2-Komplement zurücktransformiert werden.

Ist der Divisor eine Zweierpotenz 2^r, so kann die Division auch durch r arithmetische Rechtsverschiebungen des Dividenden erreicht werden. Eine arithmetische Rechtsverschiebung ändert nämlich das Vorzeichen der verschobenen Zahl nicht, halbiert jedoch ihren Betrag. Die r aus dem Dividenden hinausgeschobenen Bits geben den Rest als nichtnegative ganze Zahl an.

24.3 Dezimal-codierte vorzeichenlose ganze Zahlen

Hinter der Verwendung dezimal-codierter Zahlen steckt die Idee, daß wir als Benutzer einer Rechenanlage Zahlen meist dezimal-codiert eingeben und auch in dieser Form wieder als Ergebnis erhalten wollen. Um Konvertierungen (und bei Gleitpunktzahlen auch Konvertierungsfehler) zu vermeiden oder die Konvertierung zumindest zu vereinfachen, stellen wir die Zahlen auch intern dezimal-codiert dar.

Bei der Darstellung vorzeichenloser ganzer dezimaler Zahlen bedienen wir uns der sogenannten *BCD-Darstellung* (binary coded decimal). Jede Dezimalziffer wird separat binär in einem Nibble codiert; da ein Nibble 16 verschiedene Werte repräsentieren kann, verschenken wir bei dieser Darstellung allerdings etwa ein Drittel des Speicherplatzes. Wie bei allen Stellenwertsystemen auf dem Z80 kommt die niederwertigste Ziffer in den niederwertigsten Nibble der Folge von Nibbles, welche die Zahl darstellt. Grundsätzlich wollen wir nur Darstellungen betrachten, in denen eine gerade Anzahl von Nibbles verwendet wird; wir können dann ohne Einschränkungen Byte-Arithmetik anwenden.

Statt spezieller arithmetischer Befehle für dezimal-codierte Zahlen verfügt der Z80 über einen Anpassungsbefehl: DAA (decimal arithmetic adjust). Wird dieser Befehl unmittelbar nach einem arithmetischen Befehl für binär-codierte Zahlen ausgeführt, so wird der Inhalt des A-Registers nachträglich so modifiziert, daß er das Ergebnis dezimaler Arithmetik darstellt. Wir brauchen deshalb in die Programme aus Unterkapitel 24.1 nur an den richtigen Stellen DAA-Befehle einsetzen. Die Additions-Routine lautet damit:

```
        EXX                         ; sekundaeren Registersatz holen
        LD      B,LAENGE            ; Schleifenzaehler aufsetzen
        OR      A                   ; Uebertrag-Flag loeschen
ADD:    EXX                         ; primaeren Registersatz holen
        LD      A,(DE)              ; Byte des ersten Operanden holen
        ADC     A,(HL)              ; Byte des zweiten Operanden
                                    ; dazu addieren,
                                    ; Uebertrag einbeziehen
        DAA                         ; Ergebnis an dezimale
                                    ; Arithmetik anpassen
        LD      (BC),A              ; Byte des Ergebnisses sichern
        INC     BC                  ; auf jeweils
        INC     DE                  ; naechstes Byte
        INC     HL                  ; der Zahlen zeigen
        EXX                         ; sekundaeren Registersatz holen
        DJNZ    ADD                 ; alle Bytes der Zahlen
                                    ; bearbeiten
        EXX
        JP      C,FEHLER            ; es ist Ueberlauf aufgetreten
```

Entsprechend für die Subtraktion:

```
        EXX                         ; sekundaeren Registersatz holen
        LD      B,LAENGE            ; Schleifenzaehler aufsetzen
        OR      A                   ; Uebertrag-Flag loeschen
SUB:    EXX                         ; primaeren Registersatz holen
        LD      A,(DE)              ; Byte des ersten Operanden holen
        SBC     A,(HL)              ; Byte des zweiten Operanden
                                    ; davon subtrahieren, eventuell
                                    ; geborgtes Bit einbeziehen
        DAA                         ; Ergebnis an dezimale
                                    ; Arithmetik anpassen
        LD      (BC),A              ; Byte des Ergebnisses sichern
        INC     BC                  ; auf jeweils
        INC     DE                  ; naechstes Byte
        INC     HL                  ; der Zahlen zeigen
```

```
        EXX                     ; sekundaeren Registersatz holen
        DJNZ    SUB             ; alle Bytes der Zahlen
                                ; bearbeiten
        EXX
        JP      C,FEHLER        ; zu subtrahierende Zahl groesser
                                ; als Zahl, von der subtrahiert
                                ; werden soll
```

Bei der Multiplikation machen wir Gebrauch von einer kleinen Multiplikationstabelle, welche die Ergebnisse der Multiplikation zweier Dezimalziffern enthält (Tabelle des kleinen Einmaleins). Die Tabelle ist als Feld von Bytes aufgebaut, da das Produkt zweier Dezimalziffern nicht unbedingt durch eine Dezimalziffer dargestellt werden kann, zwei Dezimalziffern zur Darstellung aber stets genügen. Das Feld ist zweidimensional; jede Zeile enthält die Produkte einer bestimmten Ziffer des Multiplikators mit allen möglichen Ziffern des Multiplikanden.

Die Adressierung des gesuchten Feldelements erfolgt zweistufig: zunächst berechnen wir mit Hilfe der Dezimalziffer des Multiplikators (Zeilenindex) die Basisadresse der entsprechenden Zeile; diese Basisadresse entnehmen wir einem eindimensionalen Feld von Adressen. Dann verwenden wir die Dezimalziffer des Multiplikanden als Spaltenindex und gelangen so zur Adresse des gesuchten Feldelements. Folgende Routine erhält im A-Register die Dezimalziffer des Multiplikators und liefert im DE-Register die Basisadresse der entsprechenden Zeile zurück:

```
ZEILE:  PUSH    HL          ; Registerinhalt sichern
        LD      H,0         ; Zeilenindex zu Wort
        LD      L,A         ; erweitern
        ADD     HL,HL       ; Relativadresse der
                            ; Zeilenadresse berechnen
        LD      DE,BASIS    ; Basisadresse der
                            ; Adresstabelle laden
        ADD     HL,DE       ; Absolutadresse der
                            ; Zeilenadresse berechnen
        LD      E,(HL)      ; Zeilenadresse
        INC     HL          ; aus Tabelle
        LD      D,(HL)      ; entnehmen
        POP     HL          ; Register restaurieren
        RET
```

Mit der nächsten Routine beschaffen wir uns das Ergebnis der Multiplikation zweier Dezimalziffern im C-Register; wir versorgen das Unterprogramm mit der Dezimalziffer des Multiplikanden im A-Register und der Zeilenadresse im DE-Register:

```
PROD:   PUSH    HL          ; Register sichern
        LD      H,0         ; Index zu Wort
```

```
        LD      L,A         ; erweitern
        ADD     HL,DE       ; Adresse des Produkts berechnen
        LD      C,(HL)      ; Produkt holen
        POP     HL          ; Register restaurieren
        RET
```

Die Hilfstabelle hat die Form

```
BASIS:
        DEFW    PRFELD+00   ; Adresse der 0. Zeile
        DEFW    PRFELD+10   ; Adresse der 1. Zeile
        DEFW    PRFELD+20   ; Adresse der 2. Zeile
        DEFW    PRFELD+30   ; Adresse der 3. Zeile
        DEFW    PRFELD+40   ; Adresse der 4. Zeile
        DEFW    PRFELD+50   ; Adresse der 5. Zeile
        DEFW    PRFELD+60   ; Adresse der 6. Zeile
        DEFW    PRFELD+70   ; Adresse der 7. Zeile
        DEFW    PRFELD+80   ; Adresse der 8. Zeile
        DEFW    PRFELD+90   ; Adresse der 9. Zeile
```

Die Produkttabelle selbst enthält die BCD-Darstellung der Produkte:

```
PRFELD:
        DEFB    00H         ; 0 * 0
        DEFB    00H         ; 1 * 0
        :
        :
        :
        DEFB    09H         ; 3 * 3
        DEFB    12H         ; 4 * 3
        DEFB    15H         ; 5 * 3
        :

        :
        DEFB    72H         ; 8 * 9
        DEFB    81H         ; 9 * 9
```

Wir bearbeiten nun in einer Schleife die Nibbles des Multiplikators, beginnend beim höchstwertigen Nibble. Zuerst holen wir uns den Nibble und generieren aus ihm die Zeilenadresse für die Multiplikationstabelle; den Nibble selbst brauchen wir anschließend nicht mehr, alle Operationen werden über die Zeilenadresse abgewickelt. Mit Hilfe des Unterprogramms MULADD addieren wir das dem Nibble entsprechende Vielfache des Multiplikanden zum Zwischenergebnis. Das Holen des Nibbles bewerkstelligen wir durch indirekt adressiertes Rotieren eines Bytes.

Zur korrekten Versorgung des Unterprogramms MULADD bringen wir den Zeiger auf das Ergebnis ins HL'-Register.

```
        XOR     A           ; Akkumulator loeschen
        PUSH    BC          ; Zeiger auf Ergebnis sichern
        LD      BC,LAENGE   ; Zeiger hinter erstes Byte des
        ADD     HL,BC       ; Multiplikators berechnen
        EXX                 ; Zeiger auf das Ergebnis ins
        POP     HL          ; HL'-Register bringen
        LD      B,LAENGE    ; Anzahl der Bytes
                            ; des Multiplikators laden
MULTI:  EXX                 ; primaeren Registersatz holen
        LD      B,2         ; Anzahl der Nibbles je Byte
        DEC     HL          ; auf naechstes Byte des
                            ; Multiplikators zeigen
MULTIN: RLD                 ; Nibble des
                            ; Multiplikators holen
        EX      DE,HL       ; Zeiger auf Multiplikanden
        EXX                 ; holen
        LD      C,A         ; Nibble und
        PUSH    BC          ; Zaehler sichern
        CALL    ZEILE       ; Zeilenadresse berechnen
        CALL    MULADD      ; Vielfaches des Multiplikanden
                            ; zu Zwischenergebnis addieren
        POP     BC          ; Zaehler und
        LD      A,C         ; Nibble restaurieren
        EXX                 ; Zeiger auf Multiplikator
        EX      DE,HL       ; wieder beschaffen
        DJNZ    MULTIN      ; alle Nibbles eines Bytes
                            ; verarbeiten
        RLD                 ; Byte restaurieren
        EXX                 ; Zaehler beschaffen
        DJNZ    MULTI       ; alle Bytes des Multiplikators
                            ; verarbeiten
```

Zur Berechnung eines Vielfachen des Multiplikanden bilden wir sukzessive die Teilprodukte der Ziffer des Multiplikators mit den einzelnen Ziffern des Multiplikanden und addieren diese Teilprodukte an der richtigen Stelle zum Zwischenergebnis. Anschließend verschieben wir das Zwischenergebnis um einen Nibble nach links.

Die Teilprodukte sind jeweils um einen Nibble nach links gegeneinander versetzt; dies ist für das direkte Aufaddieren ziemlich hinderlich. Wir modifizieren deshalb das Verfahren geringfügig, um einen etwas glatteren Ablauf zu erzwingen. Zunächst bilden wir alle Teilprodukte der Ziffer des Multiplikators mit den niederwertigen Ziffern der Bytes des Multiplikanden;

diese Teilprodukte sind gegeneinander jeweils um ein Byte versetzt und können unter Berücksichtigung auftretender Überträge durchgängig auf das Zwischenergebnis aufaddiert werden. Als nächstes verschieben wir das Zwischenergebnis um einen Nibble nach links. Nun bilden wir alle Teilprodukte der Ziffer des Multiplikators mit den höherwertigen Ziffern der Bytes des Multiplikanden; diese sind wiederum um ein Byte gegeneinander versetzt und können wegen der vorhergehenden Verschiebung des Zwischenergebnisses direkt auf dieses aufaddiert werden.

Das Unterprogramm MULADD besteht damit aus fünf Teilen:

1. Bildung der Teilprodukte aus den niederwertigen Ziffern und Aufaddieren auf das Zwischenergebnis.
2. Verarbeitung eines eventuell dann noch vorliegenden Übertrags.
3. Linksverschiebung des Zwischenergebnisses.
4. Bildung der Teilprodukte aus den höherwertigen Ziffern und Aufaddieren auf das Zwischenergebnis.
5. Verarbeitung eines eventuell dann noch vorliegenden Übertrags.

Beim Eintritt in MULADD müssen folgende Zeiger übergeben werden: DE ist die Zeilenadresse in der Multiplikationstabelle, HL zeigt auf das Zwischenergebnis, HL' zeigt auf den Multiplikanden.

```
MULADD:  EXX
                                    ; Zeiger
         PUSH    HL                 ; auf
         EXX
                                    ; Multiplikanden sichern
         PUSH    HL                 ; Zeiger auf Zwischenergebnis
                                    ; sichern
         LD      B,LAENGE           ; Anzahl der Bytes des
                                    ; Multiplikanden laden
         OR      A                  ; Uebertrag-Flag loeschen
         EX      AF,AF'             ; und sichern
LSN:     EXX
                                    ; Zeiger auf Multiplikanden
                                    ; verfuegbar machen
         LD      A,(HL)             ; niederwertigen Nibble aus Byte
         AND     0FH                ; des Multiplikanden holen
         INC     HL                 ; auf naechstes Byte des
                                    ; Multiplikanden zeigen
         EXX
                                    ; Zeiger auf Multiplikanden
                                    ; sichern
         CALL    PROD               ; Teilprodukt im C-Register
                                    ; berechnen
         EX      AF,AF'             ; altes Uebertrag-Flag holen
         LD      A,(HL)             ; Byte des Ergebnisses holen
         ADC     A,C                ; Teilprodukt auf Ergebnis
```

```
                                        ; aufaddieren
        DAA                             ; Korrektur fuer dezimale
                                        ; Arithmetik
        LD      (HL),A                  ; Byte des Ergebnisses
                                        ; abspeichern
        EX      AF,AF'                  ; Uebertrag-Flag sichern
        INC     HL                      ; auf naechstes Byte des
                                        ; Ergebnisses zeigen
        DJNZ    LSN                     ; alle niederwertigen Nibbles
                                        ; des Multiplikanden bearbeiten
        LD      B,LAENGE                ; Restlaenge des Ergebnisses
                                        ; laden
        EX      AF,AF'                  ; altes Uebertrag-Flag holen
LSREST: LD      A,(HL)                  ; Uebertrag
        ADD     A,1                     ; auf
        DAA                             ; Ergebnis
        LD      (HL),A                  ; aufaddieren
        JP      NC,LSFERT               ; kein weiterer Uebertrag
        INC     HL                      ; auf naechstes Byte des
                                        ; Ergebnisses zeigen
        DJNZ    LSREST                  ; Uebertrag durch restliches
                                        ; Ergebnis durchziehen
LSFERT: POP     HL                      ; Zeiger auf Ergebnis
        PUSH    HL                      ; restaurieren und sichern
        LD      B,2*LAENGE              ; Laenge des Ergebnisses laden
        XOR     A                       ; A-Register loeschen
SCHIEB: RLD                             ; Nibble des Ergebnisses
                                        ; rotieren
        INC     HL                      ; auf naechstes Byte des
                                        ; Ergebnisses zeigen
        DJNZ    SCHIEB                  ; gesamtes Ergebnis um einen
                                        ; Nibble linksverschieben
        POP     HL                      ; Zeiger auf das Ergebnis
                                        ; restaurieren
        EXX                             ; Zeiger auf
        POP     HL                      ; den Multiplikanden
        PUSH    HL                      ; restaurieren
        EXX                             ; und sichern
        PUSH    HL                      ; Zeiger auf das Ergebnis sichern
        INC     HL                      ; Teilprodukte aus den
                                        ; hoeherwertigen Nibbles
                                        ; werden um ein Byte versetzt
                                        ; aufaddiert
```

```
        LD      B,LAENGE    ; Anzahl der Bytes des
                            ; Multiplikanden laden
        OR      A           ; Uebertrag-Flag loeschen
        EX      AF,AF'      ; und sichern
MSN:    EXX                 ; Zeiger auf Multiplikanden
                            ; verfuegbar machen
        LD      A,(HL)      ; hoeherwertigen Nibble
        SRL     A           ; aus Byte
        SRL     A           ; des
        SRL     A           ; Multiplikanden
        SRL     A           ; holen
        INC     HL          ; auf naechstes Byte des
                            ; Multiplikanden zeigen
        EXX                 ; Zeiger auf Multiplikanden
                            ; sichern
        CALL    PROD        ; Teilprodukt im C-Register
                            ; berechnen
        EX      AF,AF'      ; altes Uebertrag-Flag holen
        LD      A,(HL)      ; Byte des Ergebnisses holen
        ADC     A,C         ; Teilprodukt auf Ergebnis
                            ; aufaddieren
        DAA                 ; Korrektur fuer dezimale
                            ; Arithmetik
        LD      (HL),A      ; Byte des Ergebnisses
                            ; abspeichern
        EX      AF,AF'      ; Uebertrag-Flag sichern
        INC     HL          ; auf naechstes Byte des
                            ; Ergebnisses zeigen
        DJNZ    MSN         ; alle hoeherwertigen Nibbles
                            ; des Multiplikanden bearbeiten
        LD      B,LAENGE-1  ; Restlaenge des Ergebnisses
                            ; laden
        EX      AF,AF'      ; altes Uebertrag-Flag holen
MSREST: LD      A,(HL)      ; Uebertrag
        ADD     A,1         ; auf
        DAA                 ; Ergebnis
        LD      (HL),A      ; aufaddieren
        JP      NC,MSFERT   ; kein weiterer Uebertrag
        INC     HL          ; auf naechstes Byte des
                            ; Ergebnisses zeigen
        DJNZ    MSREST      ; Uebertrag durch restliches
                            ; Ergebnis durchziehen
MSFERT: POP     HL          ; Zeiger auf das Ergebnis
```

```
                                    ; restaurieren
        EXX                         ; Zeiger auf den
        POP         HL              ; Multiplikanden
        EXX                         ; restaurieren
        RET
```

Eine Multiplikation mit einer Zehnerpotenz 10^r führen wir durch r-fache Linksverschiebung des Multiplikanden um je einen Nibble durch.

Bei der Division zweier ganzer vorzeichenloser dezimal-codierter Zahlen gelten die Bemerkungen aus Unterkapitel 24.1. Die Algorithmen stimmen nahezu überein. Wesentlicher Unterschied ist die Tatsache, daß jede Ziffer des Quotienten auch einen Wert größer als eins annehmen kann, so daß zur Bestimmung der Ziffer eventuell mehrere Tests und Subtraktionen des Divisors nötig sind; auch wird statt eines Bits ein ganzer Nibble aus dem Dividenden herausgeschoben, der bei der Subtraktion miteingeschlossen werden muß.

Das modifizierte Programm lautet:

```
        PUSH        HL              ; Zeiger auf Divisor sichern
        PUSH        BC              ; Zeiger auf
        EXX                         ; Quotienten in
        POP         HL              ; sekundaeres Register bringen
        LD          B,LAENGE        ; Laenge des Quotienten laden
QULOES: LD          (HL),0          ; Byte des Quotienten loeschen
        INC         HL              ; auf naechstes Byte
                                    ; des Quotienten zeigen
        DJNZ        QULOES          ; gesamten Quotienten loeschen
        POP         DE              ; Zeiger auf Divisor holen
        LD          HL,LAENGE-1     ; Zeiger auf MSB
        ADD         HL,DE           ; des Divisors berechnen
        XOR         A               ; Anzahl der Tests ruecksetzen
NORM:   INC         A               ; Anzahl der Tests um 1 erhoehen
        INC         (HL)            ; hoechsten Nibble des Divisors
        INC         (HL)            ; testen
        JP          NZ,NORMIE       ; Divisor ist normiert
        EX          DE,HL           ; Zeiger auf Divisor holen
        EX          AF,AF'          ; Zaehler sichern
        XOR         A               ; Akkumulator loeschen
        LD          B,LAENGE        ; Laenge des Divisors laden
        PUSH        HL              ; Zeiger auf Divisor sichern
SCHIDR: RLD                         ; Byte des Divisors
                                    ; linksverschieben
        INC         HL              ; auf naechstes Byte
                                    ; des Divisors zeigen
        DJNZ        SCHIDR          ; gesamten Divisor
```

```
                                        ; linksverschieben
        POP     HL                      ; Zeiger auf Divisor restaurieren
        EX      DE,HL                   ; und sichern
        EX      AF,AF'                  ; Zaehler holen
        CP      2*LAENGE                ; Anzahl der Tests pruefen
        JP      NZ,NORM                 ; Normierung fortsetzen
        JP      FEHLER                  ; Fehler: Division durch Null
NORMIE: EXX                             ; primaeren Registersatz holen
        PUSH    BC                      ; Zeiger auf Quotienten sichern
        LD      C,A                     ; Anzahl der Divisionsschritte
                                        ; kopieren
        EXX                             ; Anzahl der
        LD      C,A                     ; Divisionsschritte
        EXX                             ; sichern
        XOR     A                       ; hoechsten Nibble des Dividenden
                                        ; loeschen
        EX      AF,AF'                  ; hoechsten Nibble des
                                        ; Dividenden sichern
QUSCHI: XOR     A                       ; Akkumulator loeschen
        LD      B,LAENGE                ; Laenge des Quotienten
        EX      (SP),HL                 ; Zeiger auf Quotienten holen,
                                        ; Zeiger auf Divisor sichern
        PUSH    HL                      ; Zeiger auf Quotienten sichern
SCHIQU: RLD                             ; Byte des Quotienten
                                        ; linksverschieben
        INC     HL                      ; auf naechstes Byte
                                        ; des Quotienten zeigen
        DJNZ    SCHIQU                  ; gesamten Quotienten
                                        ; linksverschieben
        POP     HL                      ; Zeiger auf Quotienten holen
        EX      (SP),HL                 ; Zeiger auf Divisor holen,
                                        ; Zeiger auf Quotienten sichern
        EXX                             ; Zaehler fuer
        LD      B,0                     ; Ziffernwert
        EXX                             ; loeschen
DIVI:   OR      A                       ; Uebertrag-Flag loeschen
        LD      B, LAENGE               ; Laenge des Dividenden
        PUSH    HL                      ; Zeiger auf Divisor und
        PUSH    DE                      ; Zeiger auf Dividenden sichern
SUBTRA: LD      A,(DE)                  ; Byte des
        SBC     A,(HL)                  ; Divisors von Byte
        DAA                             ; des Dividenden
        LD      (DE),A                  ; subtrahieren
```

```
        INC     DE      ; auf naechstes Byte von
        INC     HL      ; Divisor und Dividend zeigen
        DJNZ    SUBTRA  ; Divisor von Dividend
                        ; subtrahieren
        POP     DE      ; Zeiger auf Dividend und
        POP     HL      ; Divisor restaurieren
        JP      NC,QUOT1 ; kein Ueberlauf, Subtraktion
                        ; erfolgreich
        EX      AF,AF   ; hoechsten Nibble des Dividenden
                        ; holen
        OR      A       ; Nibble auf Null testen
        JP      Z,QUOTO ; Ueberlauf bei Subtraktion
                        ; rueckgaengig machen
        DEC     A       ; Uebertrag von hoechstem Bit
                        ; des Dividenden abziehen
        EX      AF,AF   ; hoechsten Nibble des
                        ; Dividenden sichern
QUOT1:  EXX             ; Ziffernwert
        INC     B       ; um 1
        EXX             ; erhoehen
        JP      DIVI    ; weiter ausdividieren
QUOTO:  EX      AF,AF   ; hoechsten Nibble des Dividenden
                        ; sichern
        OR      A       ; Uebertrag-Flag loeschen
        LD      B,LAENGE ; Laenge des Dividenden laden
        PUSH    HL      ; Zeiger auf Divisor und
        PUSH    DE      ; Zeiger auf Dividend sichern
ADDIER: LD      A,(DE)  ; Byte des
        ADC     A,(HL)  ; Divisors zu Byte
        DAA             ; des Dividenden
        LD      (DE),A  ; addieren
        INC     DE      ; auf naechstes Byte von
        INC     HL      ; Divisor und Dividend zeigen
        DJNZ    ADDIER  ; Divisor zu Dividend addieren
        POP     DE      ; Zeiger auf Dividend und
                        ; Divisor restaurieren
        EXX             ; Ziffernwert
        LD      A,B     ; holen
        EXX             ; und
        EX      (SP),HL ; in
        OR      (HL)    ; niederwertigsten
        LD      (HL),A  ; Nibble des Quotienten
        EX      (SP),HL ; eintragen
```

```
DDSCHI:  EX      AF,AF'        ; hoechsten Nibble des Dividenden
                               ; holen, dieser hat Wert Null
         LD      B,LAENGE      ; Laenge des Dividenden laden
         EX      DE,HL         ; Zeiger auf
                               ; Dividend holen
         PUSH    HL            ; und sichern
SCHIDD:  RLD                   ; Byte des Dividenden
                               ; linksverschieben
         INC     HL            ; auf naechstes Byte
                               ; des Dividenden zeigen
         DJNZ    SCHIDD        ; gesamten Dividenden
                               ; linksverschieben
         EX      AF,AF'        ; hoechsten Nibble des Dividenden
                               ; sichern
         POP     HL            ; Zeiger auf Dividend
                               ; restaurieren
         EX      DE,HL         ; Zeiger auf Divisor holen
         DEC     C             ; Anzahl der Divisionsschritte
                               ; um eins vermindern
         JP      NZ,QUSCHI     ; weiteren Divisionsschritt
                               ; ausfuehren
         EX      DE,HL         ; Zeiger auf Rest holen
         PUSH    HL            ; und sichern
         LD      BC,LAENGE-1   ; Zeiger auf MSB des Rests
         ADD     HL,BC         ; berechnen
         EXX                   ; Anzahl der
         LD      A,C           ; Rechtsverschiebungen
         EXX                   ; des Divisionsrests
         LD      C,A           ; + 1 holen
         XOR     A             ; Akkumulator loeschen
         JP      TEST          ; in abweisende Schleife springen
RENORM:  LD      B,LAENGE      ; Laenge des Rests
         PUSH    HL            ; Zeiger auf MSB des Rests
                               ; sichern
RESCHI:  RRD                   ; Byte des Rests
                               ; rechtsverschieben
         DEC     HL            ; auf naechstes Byte
                               ; des Rests zeigen
         DJNZ    RESCHI        ; gesamten Rest rechtsverschieben
         POP     HL            ; Zeiger auf MSB des Rests holen
TEST:    DEC     C             ; Anzahl der restlichen
                               ; Rechtsverschiebungen des
                               ; Divisionsrests berechnen
```

```
        JP      NZ,RENORM   ; weitere Rechtsverschiebung
                            ; ausfuehren
        POP     HL          ; Zeiger auf Rest restaurieren
        EX      DE,HL       ; Zeiger tauschen
        POP     BC          ; Zeiger auf Quotienten
                            ; restaurieren
```

Eine Division durch 10^r bewerkstelligen wir durch r-Rechtsverschiebungen des Dividenden um einen Nibble; als vorderste Ziffer muß dabei eine Null eingefügt werden.

24.4 Dezimal-codierte vorzeichenbehaftete ganze Zahlen

Dezimal-codierte vorzeichenbehaftete ganze Zahlen liegen meist in Vorzeichen/Betrag-Darstellung vor. Eine wichtige Darstellung ist das sogenannte *packed BCD* nach dem IEEE-Standard (IEEE = institute of electrical and electronical engineers). Die Zahlen haben dabei eine feste Länge von 10 Bytes. Die niederwertigen 9 Bytes enthalten zusammen 18 Dezimalziffern. Das höchstwertige Byte enthält in Bit 7 das Vorzeichen, wobei wie gewohnt 0 für positives Vorzeichen, 1 für negatives Vorzeichen steht; die restlichen Bits des höchstwertigen Bytes tragen den Wert Null.

Bei Zahlen in Vorzeichen/Betrag-Darstellung werden Vorzeichen und Betrag getrennt manipuliert.

Bei der Addition zweier Zahlen haben wir zwei Fälle zu unterscheiden: besitzen beide Zahlen dasselbe Vorzeichen, so stimmt dieses mit dem Vorzeichen des Ergebnisses überein; der Betrag des Ergebnisses ist die Summe der Beträge der Operanden. Differieren die Vorzeichen, so wird zunächst der Betrag der negativen Zahl vom Betrag der positiven Zahl abgezogen. Geht dies ohne Überlauf vonstatten, so ist das Resultat positiv und besitzt als Betrag die berechnete Differenz der Beträge der Operanden. Ansonsten ist das Resultat negativ, der Betrag ist das Komplement der berechneten Differenz der Beträge der Operanden; in diesem Fall ist es am günstigsten, die Subtraktion mit vertauschten Operanden nochmals auszuführen.

Wir schaffen uns als erstes zwei Unterprogramme für die Addition und die Subtraktion der Beträge:

```
BADD:   EXX                 ; sekundaeren Registersatz holen
        LD      B,9         ; Schleifenzaehler aufsetzen
        OR      A           ; Uebertrag-Flag loeschen
ADD:    EXX                 ; primaeren Registersatz holen
        LD      A,(DE)      ; Byte des ersten Operanden holen
        ADC     A,(HL)      ; Byte des zweiten Operanden
                            ; dazu addieren,
                            ; Uebertrag einbeziehen
        DAA                 ; Ergebnis an dezimale
                            ; Arithmetik anpassen
```

```
        LD      (BC),A      ; Byte des Ergebnisses sichern
        INC     BC          ; auf jeweils
        INC     DE          ; naechstes Byte
        INC     HL          ; der Zahlen zeigen
        EXX                 ; sekundaeren Registersatz holen
        DJNZ    ADD         ; alle Bytes der Zahlen
                            ; bearbeiten
        EXX                 ; primaeren Registersatz holen
        RET
```

Entsprechend für die Subtraktion:

```
BSUB:   EXX                 ; sekundaeren Registersatz holen
        LD      B,9         ; Schleifenzaehler aufsetzen
        OR      A           ; Uebertrag-Flag loeschen
SUB:    EXX                 ; primaeren Registersatz holen
        LD      A,(DE)      ; Byte des ersten Operanden holen
        SBC     A,(HL)      ; Byte des zweiten Operanden
                            ; davon subtrahieren, eventuell
                            ; geborgtes Bit einbeziehen
        DAA                 ; Ergebnis an dezimale
                            ; Arithmetik anpassen
        LD      (BC),A      ; Byte des Ergebnisses sichern
        INC     BC          ; auf jeweils
        INC     DE          ; naechstes Byte
        INC     HL          ; der Zahlen zeigen
        EXX                 ; sekundaeren Registersatz holen
        DJNZ    SUB         ; alle Bytes der Zahlen
                            ; bearbeiten
        EXX                 ; primaeren Registersatz holen
        RET
```

Wir brauchen nun noch ein Programmstück, das die beiden Vorzeichen vergleicht und das entsprechende Unterprogramm aufruft:

```
        PUSH    HL          ; Register-
        PUSH    DE          ; inhalte
        PUSH    BC          ; sichern
        LD      BC,9        ; Laenge des Betrags in Bytes
        ADD     HL,BC       ; Zeiger auf Vorzeichen
                            ; des zweiten Operanden
        EX      DE,HL       ; Zeiger tauschen
        ADD     HL,BC       ; Zeiger auf Vorzeichen
```

```
                                        ; des ersten Operanden
          LD        A,(DE)              ; beide Vorzeichen
          CP        (HL)                ; vergleichen
          POP       BC                  ; Register
          POP       DE                  ; restaurieren
          POP       HL                  ;
          JP        Z,GLEICH            ; gleiche Vorzeichen
          JP        C,POSNEG            ; erster Operand positiv,
                                        ; zweiter Operand negativ
          EX        DE,HL               ; Operanden tauschen
POSNEG:   PUSH      BC                  ; Register-
          PUSH      DE                  ; inhalte
          PUSH      HL                  ; sichern
          CALL      BSUB                ; Differenz der Betraege bilden
          POP       HL                  ; Register
          POP       DE                  ; restaurieren
          JP        C,VERT              ; Differenz negativ,
                                        ; Operanden tauschen
          XOR       A                   ; Akku loeschen
          LD        (BC),A              ; positives Vorzeichen eintragen
          POP       BC                  ; Register restaurieren
          RET
VERT:     POP       BC                  ; Register restaurieren
          EX        DE,HL               ; Operanden tauschen
          CALL      BSUB                ; Differenz der Betraege bilden
          LD        A,80H               ; negatives Vorzeichen
          LD        (BC),A              ; eintragen
          RET
GLEICH:   EX        AF,AF'              ; gemeinsames Vorzeichen sichern
          CALL      BADD                ; Summe der Betraege bilden
          EX        AF,AF'              ; Vorzeichen holen
          LD        (BC),A              ; und
          EX        AF,AF'              ; eintragen
          JP        C,FEHLER            ; es trat Ueberlauf auf
          RET
```

Bei der Addition kann nur dann ein Überlauf auftreten, wenn beide Operanden dasselbe Vorzeichen besitzen.

Bei der Subtraktion zweier vorzeichenbehafteter dezimal-codierter ganzer Zahlen werden die Beträge addiert, falls die Vorzeichen differieren, sonst subtrahiert. Ein Überlauf kann nur bei der Addition der Beträge vorkommen:

```
          PUSH      HL                  ; Register-
          PUSH      DE                  ; inhalte
```

```
        PUSH    BC          ; sichern
        LD      BC,9        ; Laenge des Betrags in Bytes
        ADD     HL,BC       ; Zeiger auf Vorzeichen
                            ; des zweiten Operanden
        EX      DE,HL       ; Zeiger tauschen
        ADD     HL,BC       ; Zeiger auf Vorzeichen
                            ; des ersten Operanden
        LD      A,(DE)      ; beide Vorzeichen
        CP      (HL)        ; vergleichen
        POP     BC          ; Register
        POP     DE          ; restaurieren
        POP     HL          ;
        JP      NZ,VERSCH   ; verschiedene Vorzeichen
        PUSH    BC          ; Register-
        PUSH    DE          ; inhalte
        PUSH    HL          ; sichern
        EX      AF,AF'      ; gemeinsames Vorzeichen sichern
        CALL    BSUB        ; Differenz der Betraege bilden
        POP     HL          ; Register
        POP     DE          ; restaurieren
        JP      C,VERT      ; Differenz negativ,
                            ; Operanden tauschen
        EX      AF,AF'      ; gemeinsames Vorzeichen
        LD      (BC),A      ; eintragen
        POP     BC          ; Register restaurieren
        RET
VERT:   POP     BC          ; Register restaurieren
        EX      DE,HL       ; Operanden tauschen
        CALL    BSUB        ; Differenz der Betraege bilden
        EX      AF,AF'      ; gemeinsames Vorzeichen
        XOR     10000000B   ; invertieren und
        LD      (BC),A      ; eintragen
        RET
VERSCH: EX      AF,AF'      ; Vorzeichen des ersten
                            ; Operanden sichern
        CALL    BADD        ; Summe der Betraege bilden
        EX      AF,AF'      ; Vorzeichen holen
        LD      (BC),A      ; und
        EX      AF,AF'      ; eintragen
        JP      C,FEHLER    ; es trat Ueberlauf auf
        RET
```

Zur Komplementierung einer Zahl ist nur das Vorzeichen zu vertauschen.

Bei der Multiplikation zweier vorzeichenbehafteter dezimal-codierter ganzer Zahlen beachten wir, daß der Betrag des Produkts stets gleich der Beträge der beiden Operanden ist; wir können uns dabei auf die Produktbildung aus Unterkapitel 24.3 stützen. Das Vorzeichen des Produkts ist positiv, wenn die Operanden beide dasselbe Vorzeichen besitzen, sonst negativ. Die Division erfolgt nach dem gleichen Prinzip mittels der Divisionsroutine aus Unterkapitel 24.3.

Übungen

1. Schreibe ein Komplementierprogramm für vorzeichenbehaftete dezimal-codierte ganze Zahlen.

2. Schreibe eine Multiplikationsroutine für vorzeichenbehaftete dezimal-codierte ganze Zahlen. Verwende dazu die Routinen aus Unterkapitel 24.3.

25
Gleitpunktzahlen

Gleitpunktzahlen stellen eine gebräuchliche Form der Modellierung reeller Zahlen dar. Jede Gleitpunktzahl besteht aus einer Mantisse und einem Exponenten. Die Mantisse m ist eine rationale Zahl, die in einem geeigneten Stellenwertsystem mit fester Länge exakt darstellbar ist. Der Exponent e ist eine ganze Zahl fester Länge. Durch das Paar (m,e) wird die Zahl $z = m * b^e$ mit einer bestimmten Basis b dargestellt.

Man nennt eine Gleitpunktzahl normalisiert, wenn $1/b <= m < 1$ für $m <> 0$ gilt.

25.1 Gleitpunktzahlen in Binär-Codierung

Es gibt sehr viele Darstellungsmöglichkeiten für Gleitpunktzahlen; wir sehen uns einige sehr gebräuchliche davon an (alle sind normalisiert):

Der FORTRAN- oder BASIC-Standard, bisher die auf Mikrocomputern gebräuchlichste Form, kennt zwei Formen von Gleitpunktzahlen, die sich durch die Genauigkeit der Darstellung unterscheiden.

Einfach-genaue Gleitpunktzahlen (single precision) werden mit 4 Bytes dargestellt. Die drei niederwertigen Bytes enthalten die Mantisse, das höchstwertige Byte den Exponenten. Das höchstwertige Bit des Mantissenteils beinhaltet das Vorzeichen der Mantisse (und damit das Vorzeichen der Gleitpunktzahl selbst); eine Null steht für positives Vorzeichen, eine Eins für negatives Vorzeichen. Die restlichen Bits des Mantissenteils enthalten die Ziffern der binär-codierten Mantisse mit Ausnahme der direkt nach dem Binärpunkt folgenden Ziffer Eins, die unterdrückt wird. Wenn wir statt des Vorzeichens diese Eins einsetzen, so erhalten wir in den drei Bytes des Mantissenteils eine binär-codierte vorzeichenlose ganze Zahl, die das 2^{24}fache des Betrags der Mantisse angibt.

Der Exponent, der zur Basis b=2 interpretiert wird, ergibt sich aus dem Exponententeil durch Subtraktion des Werts 80H; dieser Wert heißt *Bias*. Wenn der Exponententeil den Wert

84H hat, so ist der Exponent in Wirklichkeit 4, die Mantisse ist also mit 2^4 zu multiplizieren. Der Exponent 00H zeigt an, daß die Zahl als Ganzes den Wert Null besitzt; der darstellbare Exponentenbereich ist damit −127 bis +127.

Doppelt-genaue Gleitpunktzahlen (double precision) werden durch 8 Bytes dargestellt. Alles ist wie bei einfach-genauen Gleitpunktzahlen, außer daß die 7 niederwertigen Bytes die Mantisse darstellen (die entsprechende vorzeichenlose ganze Zahl gibt das 2^{56}fache des Betrags der Mantisse an). Die Grenzen des überstrichenen Zahlbereichs stimmen fast mit denen der einfach-genauen Gleitpunktzahlen überein, der Bereich wird aber hier feiner aufgeschlüsselt.

Immer mehr an Bedeutung gewinnt der sogenannte IEEE-Standard (IEEE = institute of electrical and electronical engineers); dieser kennt drei verschiedene Formen:

Einfach-genaue Gleitpunktzahlen (short real) belegen 4 Bytes. Das höchstwertige Bit trägt das Vorzeichen der Mantisse (Codierung wie im FORTRAN-Standard), die nächsten 8 Bits den Exponenten mit Bias 127, die restlichen 23 Bits die Binärziffern der Mantisse mit unterdrückter führender Eins (diese würde unmittelbar links vom Gleitpunkt stehen). Der Exponent 0 steht für die Zahl Null; der Exponent FFH wird verwendet, um spezielle uneigentliche Zahlen (zum Beispiel »Unendlich«) zu codieren.

Doppelt-genaue Gleitpunktzahlen (long real) besitzen 11 Bits Exponent mit Bias 1023 und 52 Bits Mantissenteil, belegen also 8 Bytes; die Form entspricht der der einfach-genauen Gleitpunktzahlen.

Hoch-genaue Gleitpunktzahlen für Zwischenergebnisse (temporary real) belegen 10 Bytes; der Exponententeil besitzt 15 Bits mit Bias 16383, der Mantissenteil 64 Bits, wobei die führende Eins unmittelbar links vom Gleitpunkt mitgespeichert wird.

Eine vom FORTRAN-Standard geringfügig abweichende Form von Gleitpunktzahlen verwendet die Sprache PASCAL in der Form des Software-Pakets TURBO-PASCAL. Die Zahlen belegen 6 Bytes, von denen 5 auf die Mantisse entfallen. Der Exponententeil befindet sich nicht im höchstwertigen Byte der Zahl, sondern im niederwertigsten Byte.

Alle genannten Formen unterscheiden sich zwar bezüglich der erreichbaren Genauigkeit und der exakten Anordnung der einzelnen Teile der Zahl, nicht aber in ihrer prinzipiellen Darstellungsweise. Wir studieren die arithmetischen Operationen deshalb am Beispiel der einfach-genauen Gleitpunktzahlen des FORTRAN-Standards.

Am einfachsten sind die Operationen Multiplikation und Division durchzuführen; es gilt nämlich (für nicht unbedingt normalisierte Gleitpunktzahlen):

$$(m_1,e_1) * (m_2,e_2) = (m_1 * m_2, e_1 + e_2)$$
$$(m_1,e_1) / (m_2,e_2) = (m_1 / m_2, e_1 - e_2)$$

Wir manipulieren deshalb zunnächst die Mantissen getrennt von den Exponenten. Bei der Multiplikation holen wir als erstes die Vorzeichen der Mantissen und setzen an deren Stelle die unterdrückten führenden Einsen ein. Eine Multiplikation der beiden 3-Byte-Beträge der Mantissen – interpretiert als vorzeichenlose ganze Zahlen – liefert ein 6-Byte-Produkt, das wieder als Betrag der Mantisse des Ergebnisses interpretiert werden kann. Dabei tritt unter Umständen der Fall auf, daß die neue Mantisse nicht normalisiert ist, das heißt mit einer Null beginnt; es

kann allerdings gezeigt werden, daß höchstens eine führende Null vorkommt (vom Spezialfall der Zahl Null abgesehen).

Durch Linksschieben der Mantisse um ein Bit stellen wir nötigenfalls die Normalisierung wieder her. Die überzähligen 3 Bytes schneiden wir einfach ab, um die Mantisse wieder an das Format einer einfach-genauen Gleitpunktzahl anzupassen. Die führende Eins der Mantisse überschreiben wir mit dem neuen Vorzeichen.

Nun addieren wir die Exponenten; wenn wir normalisieren mußten, erniedrigen wir das Resultat noch um 1. Bei der Berechnung kann Exponentenüberlauf (ein zu großer Exponent) oder Exponentenunterlauf (ein zu kleiner Exponent) auftreten. Im Falle eines Exponentenüberlaufs bricht die Multiplikation mit einem Fehler ab. Bei Exponentenunterlauf setzt man das Resultat meist Null; dies sollte aber durch eine Warnung begleitet werden.

Wir zeigen nun den Multiplikationsalgorithmus, wobei wir uns für die Mantissenmultiplikation auf die in Unterkapitel 24.1 gebrachte (oder eine ähnliche) Multiplikationsroutine für vorzeichenlose ganze Zahlen stützen.

Das DE-Register zeigt wieder auf den Multiplikanden, das HL-Register auf den Multiplikator, das BC-Register auf den Speicherplatz für das Ergebnis. Wir nehmen dazu an, daß auch für die später abzuschneidenden 3 Bytes der Mantisse des Ergebnisses genügend Platz vorhanden ist.

Der Multiplikationsalgorithmus lautet nun:

```
INC     HL          ; auf MSB der Mantisse
INC     HL          ; des Multiplikators zeigen
PUSH    HL          ; Zeiger sichern
LD      A,(HL)      ; Vorzeichen holen
SET     7,(HL)      ; unterdrueckte Eins eintragen
DEC     HL          ; auf LSB der Mantisse
DEC     HL          ; des Multiplikators zeigen
EX      DE,HL       ; Zeiger tauschen
INC     HL          ; auf MSB der Mantisse
INC     HL          ; des Multiplikanden zeigen
PUSH    HL          ; Zeiger sichern
XOR     (HL)        ; neues Vorzeichen (Maske)
OR      01111111B   ; berechnen
PUSH    AF          ; und sichern
SET     7,(HL)      ; unterdrueckte Eins eintragen
DEC     HL          ; auf LSB der Mantisse
DEC     HL          ; des Multiplikanden zeigen
PUSH    BC          ; Zeiger auf Ergebnis sichern
CALL    MULT        ; Muliplikation der Mantissen
POP     HL          ; Zeiger auf Ergebnis holen
LD      BC,5        ; Zeiger auf MSB der Mantisse
ADD     HL,BC       ; des Ergebnisses berechnen
LD      C,1         ; Flag aufsetzen
```

```
        BIT     7,(HL)      ; hoechstwertiges Bit der
                            ; Mantisse testen
        JP      NZ,NORMAL   ; Mantisse ist normalisiert
        DEC     C           ; Flag loeschen
        DEC     HL          ; auf LSB
        DEC     HL          ; der Mantisse
        DEC     HL          ; des Ergebnisses zeigen
        LD      B,4         ; ueber 4 Bytes Mantisse schieben
SCHIEB: RL      (HL)        ; Mantisse
        INC     HL          ; um ein Bit
        DJNZ    SCHIEB      ; linksverschieben
        DEC     HL
NORMAL: POP     AF          ; Vorzeichen der Mantisse holen
        AND     (HL)        ; und in MSB der Mantisse
        LD      (HL),A      ; eintragen
        INC     HL          ; Zeiger auf Exponent des
                            ; Ergebnisses berechnen
        POP     DE          ; Zeiger auf Exponent
        INC     DE          ; des Multiplikanden holen
        LD      A,(DE)      ; Exponent des Multiplikanden
        SUB     80H         ; berechnen
        LD      B,A         ; und sichern
        POP     DE          ; Zeiger auf Exponent
        INC     DE          ; des Multiplikators holen
        LD      A,(DE)      ; Exponent des Multiplikators
        SUB     81H         ; minus 1 berechnen
        RR      C           ; Flag ins Uebertrag-Flag holen
        ADC     A,B         ; Exponent des Ergebnisses
                            ; berechnen
        JP      PE,FEHLER   ; Exponentenueber- oder
                            ; unterlauf
        ADD     A,80H       ; Bias addieren
        LD      (HL),A      ; Exponent des Ergebnisses
                            ; eintragen
```

Für den Fall, daß ein Operand Null ist, führen wir eine gesonderte Behandlung durch.

Bei der Division gehen wir analog vor. Das Komplementieren einer Gleitpunktzahl geschieht durch Invertieren des Vorzeichen-Bits der Mantisse.

Problematisch sind die Operationen Addition und Subtraktion. Zunächst muß denormalisiert werden, um die beiden Exponenten einander anzugleichen; die unterdrückte Eins der Mantisse muß zu diesem Zweck wieder eingesetzt werden. Wenn sich die Exponenten um mehr als die Mantissenlänge unterscheiden, setzen wir das Ergebnis stets gleich dem betragsgrößeren der beiden Operanden.

Ob eine Addition oder eine Subtraktion der Mantissen durchgeführt wird, hängt von der Gleichheit oder Ungleichheit der Vorzeichen der beiden Operanden ab: Eine Addition zweier Gleitpunktzahlen mit gleichem Vorzeichen wird durch Mantissen-Addition, mit ungleichem Vorzeichen durch Mantissen-Subtraktion durchgeführt. Eine Subtraktion zweier Gleitpunktzahlen mit gleichem Vorzeichen führt auf eine Mantissen-Subtraktion, mit ungleichem Vorzeichen auf eine Mantissen-Addition.

Bei der Addition der Mantissen kann es vorkommen, daß die neue Mantisse um ein Bit länger ist als vorher; zur Normalisierung muß dann die Mantisse um ein Bit nach rechts verschoben werden, der Exponent wird dafür um eins erhöht.

Bei der Subtraktion der Mantissen kann dagegen das Phänomen der *Auslöschung* auftreten, das darin besteht, daß beliebig viele führende Nullen entstehen; wenn die Mantissen in normalisierter Form gleich waren und die Exponenten übereinstimmten, ist das Resultat eine Null-Mantisse. Außer im Fall einer entstandenen Null normalisieren wir durch eine entsprechende Anzahl von Linksverschiebungen der Mantisse, jeweils verbunden mit einer Erniedrigung des Exponenten um eins. Auslöschung ist der Grund für große Ungenauigkeiten bei arithmetischen Berechnungen.

Wir zeigen zunächst den Entscheidungsalgorithmus für die Addition:

```
GLADD:  INC   DE          ; auf MSB der Mantisse
        INC   DE          ; des 1. Operanden zeigen
        INC   HL          ; auf MSB der Mantisse
        INC   HL          ; des 2. Operanden zeigen
        LD    A,(HL)      ; Vorzeichen des 2. Operanden
        EX    AF,AF'      ; sichern
        LD    A,(DE)      ; Vorzeichen des 1. Operanden
        XOR   (HL)        ; beide Vorzeichen vergleichen
        LD    A,(DE)      ; Vorzeichen des 1. Operanden
        JP    P,ADDIER    ; Vorzeichen gleich, addieren
        JP    SUBTRA      ; Vorzeichen verschieden,
                          ; subtrahieren
```

Der Entscheidungsalgorithmus für die Subtraktion unterscheidet sich davon nur durch die Sprungbedingung und die Inversion des Vorzeichens des 2. Operanden:

```
GLSUB:  INC   DE          ; auf MSB der Mantisse
        INC   DE          ; des 1. Operanden zeigen
        INC   HL          ; auf MSB der Mantisse
        INC   HL          ; des 2. Operanden zeigen
        LD    A,(HL)      ; invertiertes
        XOR   10000000B   ; Vorzeichen des 2. Operanden
        EX    AF,AF'      ; sichern
        LD    A,(DE)      ; Vorzeichen des 1. Operanden
        XOR   (HL)        ; beide Vorzeichen vergleichen
```

```
        LD      A,(DE)      ; Vorzeichen des 1. Operanden
        JP      M,ADDIER    ; Vorzeichen verschieden,
                            ; addieren
        JP      SUBTRA      ; Vorzeichen gleich, subtrahieren
```

Wir bringen als nächstes ein Unterprogramm, das die Denormalisierung durchführt. Falls die Exponenten sich mindestens um die Mantissenlänge unterscheiden, verzichten wir auf die Denormalisierung und kopieren den betragsgrößeren Operanden. Das Unterprogramm erhält im HL-Register einen Zeiger auf den Exponenten des zu denormalisierenden Operanden, im DE-Register einen Zeiger auf den Exponenten des anderen Operanden, im BC-Register einen Zeiger auf das LSB des Ergebnisses, im A-Register die Zahl der Rechtsverschiebungen. Auf dem Stapel befindet sich unter der Rückkehradresse das Vorzeichen des Ergebnisses, das zuvor berechnet wurde. Bei der Rückkehr aus dem Unterprogramm zeigen HL und DE auf die LSB der Operanden.

```
DENORM: DEC     DE          ; auf LSB
        DEC     DE          ; des ersten
        DEC     DE          ; Operanden zeigen
        OR      A           ; Anzahl der Verschiebungen
                            ; testen
        JP      Z,NICHTS    ; nichts zu schieben
        PUSH    BC          ; Zeiger sichern
        CP      24          ; Laenge der Mantisse in Bits
        JP      NC,KOPIE    ; ersten Operanden kopieren
        LD      B,A         ; Zaehler aufsetzen
SCHIEB: PUSH    HL
        DEC     HL
        SRL     (HL)
        DEC     HL
        RR      (HL)
        DEC     (HL)
        RR      (HL)
        POP     HL
        DJNZ    SCHIEB      ; denormalisieren
        POP     BC          ; Zeiger restaurieren
NICHTS: DEC     HL          ; auf LSB
        DEC     HL          ; des zweiten
        DEC     HL          ; Operanden zeigen
        RET
KOPIE:  POP     HL          ; Zeiger auf LSB des Ergebnisses
        EX      DE,HL       ; Zeiger tauschen
        LD      BC,4        ; Anzahl der zu kopierenden Bytes
        LDIR                ; kopieren
```

```
        EX      DE,HL       ; Zeiger tauschen
        DEC     HL          ; Zeiger auf MSB der Mantisse
        DEC     HL          ; des Ergebnisses
        POP     BC          ; Rueckkehradresse vernichten
VORZ:   POP     AF          ; Vorzeichen des Ergebnisses
                            ; holen
        OR      01111111B   ; Vorzeichen-Bit isolieren
        AND     (HL)        ; und in
        LD      (HL),A      ; Mantisse eintragen
FERTIG: NOP                 ; gemeinsamer Fortsetzungspunkt
                            ; nach fehlerfreier Ausfuehrung
```

Die Marke VORZ dient als Einsprungpunkt für das korrekte Setzen des Vorzeichens und wird von den folgenden Routinen für die Mantissenaddition und die Mantissensubtraktion benutzt.

```
ADDIER: PUSH    AF          ; Vorzeichen sichern
        SET     7,(HL)      ; unterdrueckte Eins eintragen
        EX      DE,HL       ; Zeiger tauschen
        SET     7,(HL)      ; unterdrueckte Eins eintragen
        INC     DE          ; auf Exponenten des ersten
                            ; Operanden zeigen
        INC     HL          ; auf Exponenten des zweiten
                            ; Operanden zeigen
        LD      A,(DE)      ; Differenz der
        SUB     (HL)        ; Operanden berechnen
        JP      NC,APOSIT   ; positive Differenz
        NEG                 ; Betrag der Differenzen bilden
        EX      DE,HL       ; Zeiger tauschen
APOSIT: CALL    DENORM      ; denormalisieren
        EXX                 ; sekundaeren Registersatz holen
        LD      B,3         ; Laenge der Mantisse in Bytes
        OR      A           ; Uebertrag-Flag loeschen
ADD:    EXX                 ; primaeren Registersatz holen
        LD      A,(DE)      ; Summe
        ADC     A,(HL)      ; bilden
        LD      (BC),A      ; und abspeichern
        INC     BC          ; Zeiger
        INC     DE          ; auf naechstes Byte
        INC     HL          ; richten
        EXX                 ; sekundaeren Registersatz holen
        DJNZ    ADD         ; Mantissen addieren
        EXX                 ; primaeren Registersatz holen
        LD      H,B         ; Zeiger
```

```
            LD      L,C         ; kopieren
            LD      A,(DE)      ; Exponent
            LD      (HL),A      ; kopieren
            DEC     HL          ; auf MSB der Mantisse des
                                ; Ergebnisses zeigen
            JP      NC,VORZ     ; Mantisse ist normalisiert, nur
                                ; noch Vorzeichen eintragen
            RR      (HL)        ; Mantisse
            DEC     HL          ; durch
            RR      (HL)        ; Rechtsschieben
            DEC     HL          ; wieder
            RR      (HL)        ; normalisieren
            INC     HL          ; auf Exponent
            INC     HL          ; des Ergebnisses
            INC     HL          ; zeigen
            INC     (HL)        ; Exponent um eins erhoehen
            JP      Z,UEBERL    ; Fehler: Exponentenueberlauf
            DEC     HL          ; auf MSB der Mantisse des
                                ; Ergebnisses zeigen
            JP      VORZ        ; Vorzeichen korrekt eintragen
SUBTRA:     SET     7,(HL)      ; unterdrueckte Eins eintragen
            EX      DE,HL       ; Zeiger tauschen
            SET     7,(HL)      ; unterdrueckte Eins eintragen
            EX      DE,HL
            INC     DE          ; auf Exponenten des ersten
                                ; Operanden zeigen
            INC     HL          ; auf Exponenten des zweiten
                                ; Operanden zeigen
            PUSH    AF          ; Vorzeichen des ersten
                                ; Operanden sichern
            LD      A,(DE)      ; Differenz der
            SUB     (HL)        ; Operanden berechnen
            JP      NC,SPOSIT   ; positive Differenz
            NEG                 ; Betrag der Differenz bilden
            EX      DE,HL       ; Zeiger tauschen
            INC     SP          ; gesichertes Vorzeichen
            INC     SP          ; vernichten
            EX      AF,AF'      ; Vorzeichen des zweiten
                                ; Operanden holen
            PUSH    AF          ; und sichern
            EX      AF,AF'      ; Anzahl der Verschiebungen
                                ; wiederbeschaffen
SPOSIT:     CALL    DENORM      ; denormalisieren
```

```
        EXX                     ; sekundaeren Registersatz holen
        LD      B,3             ; Laenge der Mantisse in Bytes
        OR      A               ; Uebertrag-Flag loeschen
SUB:    EXX                     ; primaeren Registersatz holen
        LD      A,(DE)          ; Differenz
        SBC     A,(HL)          ; bilden
        LD      (BC),A          ; und abspeichern
        INC     BC              ; Zeiger
        INC     DE              ; auf naechstes Byte
        INC     HL              ; richten
        EXX                     ; sekundaeren Registersatz holen
        DJNZ    ADD             ; Mantissen subtrahieren
        EXX                     ; primaeren Registersatz holen
        LD      H,B             ; Zeiger
        LD      L,C             ; kopieren
        LD      A,(DE)          ; Exponent
        LD      (HL),A          ; kopieren
        LD      B,24            ; Laenge der Mantisse in Bits
NORM:   DEC     HL              ; auf MSB der Mantisse zeigen
        BIT     7,(HL)          ; auf Normalisierung testen
        JP      NZ,VORZ         ; Mantisse ist normalisiert,
                                ; Vorzeichen korrekt eintragen
        DEC     HL              ; auf LSB der
        DEC     HL              ; Mantisse zeigen
        SLA     (HL)            ; eine
        INC     HL              ; Linksverschiebung
        RL      (HL)            ; der
        INC     HL              ; Mantisse
        RL      (HL)            ; durchfuehren
        INC     HL              ; auf Exponent zeigen
        DEC     (HL)            ; Exponent um eins reduzieren
        JP      Z,UNTERL        ; Fehler: Exponentenunterlauf
        DJNZ    NORM            ; Mantisse normalisieren
        LD      (HL),0          ; Zahl ist Null
        JP      FERTIG          ; nichts mehr zu tun
```

25.2 Gleitpunktzahlen in Dezimal-Codierung

Die Verwendung von Gleitpunktzahlen in Dezimal-Codierung ist bis jetzt nur in Spezialanwendungen zu finden, wird sich aber im Laufe der Zeit vielleicht auch in gängigen Systemen durchsetzen. Dezimal-codierte Gleitpunktzahlen besitzen eine Mantisse in BCD-Darstellung und einen binär-codierten Exponenten, der zur Basis b=10 interpretiert wird. Ausgehend vom

IEEE-Format für vorzeichenbehaftete dezimal-codierte ganze Zahlen bietet sich folgendes Format an:

Der Betrag der Mantisse besteht aus 9 Bytes (18 Ziffern), wobei keine Ziffern unterdrückt sind; der Dezimalpunkt würde vor der höchstwertigen Ziffer stehen. Die Ziffern der Mantisse belegen die niederwertigsten Bytes der Zahldarstellung. Anschließend an die Ziffern der Mantisse folgt das Vorzeichen der Mantisse in der Form des IEEE-Standard; das Vorzeichen belegt damit ein Byte. Dann folgt der Exponent, der ein Byte belegt und den Bias 127 besitzt. Der Exponent 0 zeigt wieder an, daß die gesamte Zahl Null ist.

Der Vorteil einer solchen Darstellung besteht darin, daß bei Ein- und Ausgabe von Dezimalzahlen kaum Konvertierungen vorgenommen werden müssen; insbesondere ist jede eingegebene Dezimalzahl mit hinreichend kleiner Stellenzahl ohne Rundungsfehler darstellbar (nicht so bei Binärdarstellungen). Da der Exponent zur Basis 10 gewertet wird, überstreicht er einen großen Zahlbereich (etwa von -10^{126} bis $+10^{127}$).

26
Ein-/Ausgabe-Techniken

Bisher haben wir stets nur Operationen auf Registern oder Speicherzellen durchgeführt. Ein Computer ist für uns aber nur dann von Nutzen, wenn er von uns Daten übernehmen und uns Ergebnisse liefern kann. Wir benötigen also noch zusätzlich Befehle für die Ein-/Ausgabe.

26.1 Allgemeines zur Ein-/Ausgabe

Ein-/Ausgabe (engl. input/output) wird häufig durch E/A (engl. I/O) abgekürzt. Um die E/A zu realisieren, muß der Prozessor hardwaremäßig mit peripheren Bausteinen oder Schaltungen verbunden sein. Es gibt dazu prinzipiell zwei Möglichkeiten: Die Peripherie kann so geschaltet sein, daß sie durch Befehle erreichbar ist, mit denen sonst Speicherzellen bearbeitet werden; dies nennt man *Speicher-orientierte* E/A (engl. memory mapped I/O). Für den Prozessor ist nicht ersichtlich, ob er auf Speicherzellen oder Speicher-adressierten externen Geräten arbeitet. Die zweite Art der Kommunikation mit externen Geräten geschieht durch spezielle E/A-Befehle; diese wirken auf sogenannte *Ports* (das sind logische E/A-Kanäle), weshalb man diese Art der E/A als *Port-adressierte* E/A (engl. port driven I/O) bezeichnet. Für die externe Hardware ist der Unterschied zwischen speicher-adressierter E/A und port-adressierter E/A nahezu belanglos. Im Falle des Z80 ist der Datenverkehr zwischen Prozessor und Peripherie auf der Seite des Prozessors stets Byte-orientiert.

Im Unterschied zu allen bisher besprochenen Problemen kann bei der E/A ein bestimmtes zeitliches Verhalten der Software erforderlich sein. Wird der Z80 als Steuerung einer Peripheriekarte eingesetzt (zum Beispiel in einem Plotter), so kann nach Ausgabe eines Signals an die angeschlossene Hardware nicht sofort mit einer Reaktion gerechnet werden, da insbesondere die mechanischen Komponenten träge sind. Umgekehrt kann es erforderlich sein, daß der Prozessor auf ein gelesenes Signal möglichst schnell reagiert. Aus Anhang B und der Taktfrequenz des Z80 können die Laufzeiten von Befehlen - und damit von Programmstücken - ermittelt werden.

Ein externes Gerät kann durch mehrere Ports oder Speicheradressen repräsentiert sein, die verschiedene Funktionen übernehmen. Häufig wird ein Status-Port vorhanden sein, an dem der Zustand des externen Geräts ablesbar ist. Über Daten-Ports werden Daten von der Peripherie übernommen oder an die Peripherie übergeben. Kontroll-Ports dienen zur Steuerung externer Geräte.

26.2 Speicher-adressierte Ein-/Ausgabe

Die speicher-adressierte E/A bietet den Vorteil, daß mit dem Gerät fiktiv wie mit einem kleinen Speicherbereich (interpretiert als Verbund) gearbeitet werden kann. Dies bedeutet, daß zum Beispiel mehrere gleichartige Geräte sich nur durch die Adressen, auf die sie gelegt sind, unterscheiden. Zur Bearbeitung können damit Indexregister oder Datenadreßregister herangezogen werden. Einige der Speicherbefehle sind auch schneller als die eigentlichen E/A-Befehle, die wir im nächsten Unterkapitel kennenlernen werden. Ein besonders extremer Fall liegt vor, wenn dem externen Gerät nicht einige wenige Adressen zugeordnet sind, sondern ein großer zusammenhängender Adreßbereich; dies trifft typischerweise auf Bildschirme zu. Mit Hilfe von Blocktransferbefehlen können wir dann sehr viele Daten in kurzer Zeit zwischen Prozessor und externem Gerät austauschen.

Wir stellen uns einen Bildschirm mit m Zeilen und n Spalten vor. In jeder Zeile steht pro Spalte genau ein ASCII-Zeichen. Wir ordnen dem Bildschirm einen Speicherbereich von m*n Bytes zu, die den Positionen der Zeichen auf dem Bildschirm entsprechen. Sobald ein Byte in einen Speicherplatz des Bildschirmspeichers geschrieben wird, erzeugt ein spezieller Baustein - der Bildschirm-Kontroller - das entsprechende Zeichen am Bildschirm.

Wir können nun vorgefertigte »Bildschirm-Seiten« im Speicher aufbewahren und mit Hilfe eines LDIR-Befehls schlagartig auf den Bildschirm bringen. Nehmen wir an, daß unser Bildschirm 16 Zeilen und 64 Spalten besitzt; der Bildschirm-Speicher soll bei Adresse 3C00H beginnen. Wenn die vorgefertigte Seite ab Adresse 7800H im Speicher steht, so lautet das Programmstück zum Sichtbarmachen der Seite:

```
LD      HL,7800H    ; Adresse der Seite
LD      DE,3C00H    ; Adresse des Bildschirmspeichers
LD      BC,16*64    ; Anzahl der Zeichen
                    ; des Bildschirms
LDIR                ; Seite zeigen
```

Bei den meisten Systemen kann der Bildschirm-Speicher auch vom Prozessor gelesen werden. Wir bringen den Inhalt des Bildschirm-Speichers zwecks Analyse in den Speicherbereich ab Adresse 7400H:

```
LD      HL,3C00H    ; Adresse des Bildschirmspeichers
LD      DE,7400H    ; Adresse der Seite
LD      BC,16*64    ; Anzahl der Zeichen
```

```
        LDIR                    ; des Bildschirms
                                ; Seite abspeichern
```

Folgende Routine löscht die i-te Zeile des Bildschirms durch Überschreiben mit Leerzeichen (i steht im A-Register und wird ab Null gezählt):

```
        LD      DE,3C00H        ; Adresse des Bildschirmspeichers
        LD      H,0             ; Index i zu
        LD      L,A             ; Wort erweitern
        LD      B,6             ; 2^6 = 64 = Zahl der Zeichen
                                ; pro Zeile
DOPPEL: ADD     HL,HL           ; Inhalt des HL-Registers
        DJNZ    DOPPEL          ; verdoppeln
        ADD     HL,DE           ; Adresse der i-ten Zeile
                                ; berechnen
        LD      (HL),' '        ; erstes Zeichen der Zeile
                                ; loeschen
        LD      D,H             ; HL-Register ins
        LD      E,L             ; DE-Register kopieren
        INC     DE              ; auf naechstes zu loeschendes
                                ; Zeichen zeigen
        LD      BC,63           ; Anzahl der noch zu loeschenden
                                ; Zeichen
        LDIR                    ; Rest der Zeile loeschen
```

Das Ansprechen des Bildschirms über Speicheradressen macht Operationen wie das Hochrollen des Bildschirms oder das Hin- und Herschieben von Teilen des Bildschirms einfach.

Bei der Tastatur liegt manchmal ein ähnliches Problem zugrunde. Die Tasten werden als rechteckiges Schema angeordnet; jeder Taste entspricht wieder genau ein Speicherplatz des Tastatur-Speichers. Im Gegensatz zu einem Zeichen auf dem Bildschirm kann eine Taste nur zwei Zustände annehmen. Es genügt zur Darstellung also ein Bit pro Taste. Wir können bei einer solchen Tastatur auch prüfen, ob mehrere Tasten gleichzeitig gedrückt sind und welche das sind. So kann man bestimmten Tastenkombinationen Sonderfunktionen unterlegen.

Kommen wir nun zu einem speicher-adressierten Gerät mit Status: dem parallelen Drucker. Einer Drucker-Schnittstelle kann ein ASCII-Zeichen übergeben werden, das diese an den Drucker weiterleitet, falls er bereit ist, ein Zeichen zu übernehmen und zu drucken. Hinderungsgründe könnten zum Beispiel sein:

- Kein Drucker angeschlossen
- Drucker nicht eingeschaltet
- Kein Papier im Drucker
- Drucker ist gerade mit Ausgabe eines Zeichens beschäftigt

Man ordnet dem Drucker nun eine Speicherzelle zu, aus welcher der Status des Druckers entnommen werden kann; jeder möglichen Störung entspricht dabei ein bestimmtes Bit. Die Bedeutung der einzelnen Bits könnte zum Beispiel sein:

Bit 7	Drucker beschäftigt (busy)
Bit 6	Kein Papier im Drucker (out of paper)
Bit 5	Drucker nicht selektiert (device not selected)

Durch Lesen des Status kann festgestellt werden, ob der Drucker zur Ausgabe bereit ist und wenn nicht, wo der Fehler zu suchen ist.

Man kann nun dieselbe Adresse benutzen, um ein Zeichen an die Drucker-Schnittstelle zu übergeben. In der einen Richtung erscheint die Speicheradresse also als Status-Port, in der anderen als Daten-Port.

Folgende Routine gibt ein im D-Register stehendes ASCII-Zeichen auf den Drucker aus, sobald dieser bereit ist (zugeordnete Adresse = 37E8H):

```
        LD      E,11100000B  ; Maske fuer Status
        LD      HL,37E8H     ; Adresse des Drucker-Speichers
WARTE:  LD      A,(HL)       ; Drucker-Status holen
        AND     E            ; irrelevante Bits wegmaskieren
        JP      NZ,WARTE     ; Status zeigt Fehler an
        LD      (HL),D       ; Zeichen an Drucker ausgeben
```

Es kann sein, daß diese Routine zeitkritisch ist, da der Status des Druckers ja nur dessen momentanen Zustand wiedergibt; wartet man mit der Ausgabe des Zeichens nach Auslesen des Status zu lange, so hat sich der Zustand (und natürlich auch der Status) möglicherweise bereits geändert.

Ein wesentlicher Nachteil der speicher-adressierten E/A ist die Zerstückelung des Hauptspeichers; die externen Geräten zugeteilten Adressen können ja nicht auch noch mit RAM oder ROM belegt sein. Als Folge entstehen Einschränkungen bezüglich Lage und Länge von Daten und Programm. Legt man also auf einen zusammenhängenden Speicher mit 64 KByte Adreßraum großen Wert, so muß man zu Port-adressierter E/A übergehen.

Übungen

1. Schreibe eine Routine zum Löschen der j-ten Spalte eines Bildschirms.

2. Schreibe eine Routine, die den Status des Druckers analysiert und abbricht, wenn ein echter Fehler (kein Papier, Gerät nicht selektiert) vorliegt; ansonsten soll mit der Ausgabe eines Zeichens gewartet werden, bis der Drucker nicht mehr aktiv ist.

26.3 Port-adressierte Ein-/Ausgabe

Der Z80 verfügt über spezielle E/A-Befehle, mit denen man 256 Ports adressieren kann. Jeder Port trägt eine Adresse, die aus 8 Bits besteht. Wir setzen unser Beispiel aus dem vorangegangenen Unterkapitel fort und ordnen dem parallelen Drucker den Port 62H zu. Zum Lesen aus einem Port bedienen wir uns des Befehls IN (in):

```
        IN      A,(62H)     ; Inhalt von Port 62H
                            ; ins A-Register bringen
```

Dieser Befehl schreibt uns den Status des Druckers ins A-Register.

Die Ausgabe des A-Registers auf einen Port erfolgt mit Hilfe des Befehls OUT (out):

```
        OUT     (62H),A     ; Inhalt des A-Registers
                            ; auf Port 62H ausgeben
```

Auch Ports können indirekt adressiert werden, und zwar durch das C-Register. Das Programmstück

```
        LD      C,62H       ; Port-Adressregister mit
                            ; Port-Adresse laden
        LD      (C),D       ; D-Register auf den Port
                            ; ausgeben, dessen Port-Adresse
                            ; im C-Register steht
```

gibt den Inhalt des D-Registers auf den Port 62H aus. Während bei der direkten Port-Adressierung (Port-Adresse steht im Befehl) stets das A-Register die Daten aufnimmt oder liefert, kann bei indirekter Port-Adressierung über das C-Register jedes der 8-Bit-Register A, B, C, D, E, H, L die Daten liefern oder aufnehmen, zum Beispiel:

```
        LD      C,62H       ; Port-Adressregister mit
                            ; Port-Adresse laden
        IN      L,(C)       ; Inhalt des Ports, dessen
                            ; Port-Adresse im C-Register
                            ; steht, ins L-Register bringen
```

Wir formulieren nun unsere Ausgabe-Routine mit Hilfe von E/A-Befehlen:

```
        LD      E,11100000B ; Maske fuer Status
        LD      C,62H       ; Port-Adressregister mit
                            ; Port-Adresse laden
WARTE:  IN      A,(C)       ; Drucker-Status holen
        AND     E           ; irrelevante Bits wegmaskieren
```

```
        JP      NZ,WARTE    ; Status zeigt Fehler an
        OUT     (C),D       ; Zeichen an Drucker ausgeben
```

Natürlich wäre es auch möglich, dem Daten-Port eine vom Status-Port verschiedene Adresse zuzuordnen, zum Beispiel die Adresse 63H. Dies ist sinnvoll, falls wir das zuletzt ausgegebene Byte wieder einlesen wollen. Das Beispiel lautet dann:

```
        LD      E,11100000B ; Maske fuer Status
WARTE:  IN      A,(62H)     ; Drucker-Status holen
        AND     E           ; irrelevante Bits wegmaskieren
        JP      NZ,WARTE    ; Status zeigt Fehler an
        LD      A,D         ; Zeichen ins A-Register holen
        OUT     (63H),A     ; Zeichen an Drucker ausgeben
```

Nun kommt eine kleine Überraschung: Die 65 536 Ports des Z80!

Bei Verwendung indirekter Port-Adressierung legt der Z80 den Wert des BC-Registers auf den Adreßbus; dies bedeutet, daß bei entsprechender hardwaremäßiger Auslegung des Systems auch das BC-Register als Port-Adreßregister verwendet werden kann, und daß jedem 16-Bit-Wert ein Port entsprechen könnte. In der Praxis braucht man nicht so viele Ports; wählt man allerdings die Portadressen 0001H, 0002H, 0004H, 0008H, ..., 8000H, so hat man 16 Ports zur Verfügung, die hardwaremäßig äußerst einfach realisiert werden können, was den Aufbau eines Computersystems billiger und sicherer werden läßt (keine Port-Adreß-Decodierung notwendig). Der Aufwand bei der Software ist dafür etwas höher als normal.

Manche E/A-Geräte (disk, high speed link) sind in der Lage, relativ große Datenmengen in kurzer Zeit zu liefern oder abzuholen. Dem trägt der Z80 durch einen Satz von Block-E/A-Befehlen Rechnung. Die Daten werden dabei stets zwischen einem Port, dessen Port-Adresse im C-Register steht, und einem zusammenhängenden Speicherbereich, auf den das HL-Register zeigt, ausgetauscht. Die E/A-Operation kann den Speicherbereich aufsteigend oder absteigend bearbeiten. Die Länge des Speicherbereichs wird durch den Wert des B-Registers gegeben. Ein Datenblock kann damit maximal 256 Bytes lang sein.

Die formale Beschreibung des Befehls **OTIR** (out, increment and repeat) sieht folgendermaßen aus:

```
wiederhole
            [<C>] <- <(<HL>)>
            HL <- <HL> + 1
            B <- <B> - 1
bis         <B> = 0
```

Wollen wir beispielsweise einen Datenblock von 128 Bytes, der ab Adresse 6980H im Speicher steht, aufsteigend auf den Port 47H ausgeben, so schreiben wir folgendes Programmstück:

```
        LD      C,47H       ; Port-Adresse
        LD      HL,6980H    ; Block-Adresse
```

```
        LD      B,128       ; Blocklaenge
        OTIR                ; Block auf Port ausgeben
```

Ist der Block dagegen 256 Bytes lang, so würde das Programmstück lauten:

```
        LD      C,47H       ; Port-Adresse
        LD      HL,6980H    ; Block-Adresse
        LD      B,0         ; Blocklaenge = 256
        OTIR                ; Block auf Port ausgeben
```

Wollen wir umgekehrt einen Block von 80 Bytes von Port 69H in einen Speicherbereich lesen, der bei 795AH beginnt, so schreiben wir folgendes Programmstück:

```
        LD      C,69H       ; Port-Adresse
        LD      HL,795AH    ; Block-Adresse
        LD      B,80        ; Blocklaenge
        INIR                ; Block von Port lesen
```

Die entsprechenden Block-E/A-Befehle für absteigende Bearbeitung lauten **OTDR** (out, decrement and repeat) und **INDR** (in, decrement and repeat).

Es gibt auch Block-E/A-Befehle, die (in Analogie zu Befehlen wie LDI) keine Wiederholung beinhalten und dadurch ein zwischenzeitliches Prüfen des Geräte-Status ermöglichen. Der Befehl **OUTI** (out and increment) besitzt zum Beispiel folgende formale Beschreibung:

```
[<C>] <- <(<HL>)>
HL <- <HL> + 1
B <- <B> - 1
```

Wenn wir einen Datenblock mit 80 Zeichen, der ab Adresse 95B0H beginnt, auf einen parallelen Drucker mit gemeinsamem Status- und Daten-Port (Port-Adresse 62H) ausgeben wollen, und der Status so wie oben zu interpretieren ist, können wir folgende Routine verwenden:

```
        LD      E,11100000B ; Maske fuer Status
        LD      C,62H       ; Port-Adresse
        LD      HL,95B0H    ; Block-Adresse
        LD      B,80        ; Blocklaenge
WARTE:  IN      A,(C)       ; Drucker-Status holen
        AND     E           ; irrelevante Bits wegmaskieren
        JP      NZ,WARTE    ; Status zeigt Fehler an
        OUTI                ; Zeichen an Drucker ausgeben
        JP      NZ,WARTE    ; gesamten Block
                            ; auf Port ausgeben
```

Weitere Block-E/A-Befehle sind **OUTD** (out and decrement), **INI** (in and increment) und **IND** (in and decrement).

Die genaue Funktion aller Block-E/A-Befehle kann aus Anhang B entnommen werden.

Übungen

1. Schreibe ein Unterprogramm, das auch Datenblocke ausgeben kann, die länger als 256 Bytes sind.

2. Modifiziere das letzte Beispiel dieses Unterkapitels so, daß abgebrochen wird, wenn ein tatsächlicher Fehler vorliegt; gewartet wird nur, solange der Drucker busy ist.

26.4 Simultanes Bedienen mehrerer Ein-/Ausgabe-Geräte

Es kommt manchmal vor, daß mehrere Eingabe-Geräte gleichzeitig an den Computer angeschlossen sind, zum Beispiel eine Tastatur, ein Graphik-Tablett und eine Maus. Diese Geräte müssen wir dann gleichzeitig im Auge behalten, das heißt ihren Status periodisch prüfen (engl. polling). Ändert sich der Status eines der Eingabe-Geräte, so bedeutet dies normalerweise, daß Daten von diesem Gerät abgeholt werden sollen; ob dies so ist, muß aber durch Analyse des Status erst festgestellt werden.

Die Reaktion auf eine Eingabe an einem der verschiedenen Eingabe-Geräte soll möglichst schnell erfolgen; dies kann auf Probleme treffen, wenn viele Eingabe-Geräte angeschlossen sind, was typischerweise auf Systeme zum Messen physikalischer Größen zutrifft (zum Beispiel Echtzeit-Systeme zur Steuerung von Maschinen). Man muß die Routinen zum Auslesen und Analysieren des Status dann so schreiben, daß sie möglichst wenig Zeit benötigen; die Programme werden dadurch meist wesentlich länger und auch schwieriger zu durchschauen.

Wenn neben der Überwachung externer Geräte auch noch ständig Berechnungen ausgeführt werden sollen, so muß ein Modus gefunden werden, die Berechnungen zeitweilig zu suspendieren und in dieser Zeit die Geräte zu beobachten. Solche Programme sind äußerst zeitkritisch; die Analyse ihres Laufzeitverhaltens ist sehr komplex. Man wählt deshalb meist den Weg über Unterbrechungen; Näheres zu diesem Thema im nächsten Kapitel!

27
Unterbrechungen

Eine *Unterbrechung* (engl. interrupt) ist ein von einem äußeren Signal ausgelöster Eingriff in den Ablauf eines Programms; eine Unterbrechung führt zur zwischenzeitlichen Ausführung einer *Unterbrechungs-Behandlungs-Routine* (engl. interrupt service routine) – kurz Unterbrechungsroutine genannt nach deren Bearbeitung das unterbrochene Programm fortgesetzt wird.

27.1 Das Unterbrechungskonzept

Bei den Ein-/Ausgabe-Techniken sind wir auf das Problem gestoßen, daß das Bedienen eines oder mehrerer (konkurrierender) externer Geräte – insbesondere mit unregelmäßigem Zeitverhalten – nur mit großem Aufwand zur Überwachung der Geräte durchführbar ist; die Steuerung der Programmabläufe wird kompliziert, worunter die Verständlichkeit und Fehlersicherheit der Programme entschieden leidet.

Das Konzept der Unterbrechungen schafft in dieser Situation gewisse Erleichterungen, die insbesondere zu einer sauberen Trennung zwischen dem aktiven Programm und den verschiedenen *Geräte-Treibern* (Unterprogrammen zur Geräte-Steuerung) führen.

Ein Unterbrechungswunsch wird dem Z80 von einem externen Gerät durch ein spezielles Steuersignal mitgeteilt. Nach Bearbeitung eines Maschinenbefehls beziehungsweise nach jeder Transport- oder Vergleichsoperation in Blockbefehlen prüft der Prozessor, ob ein Unterbrechungswunsch besteht (und ob dieser zugelassen ist). In diesem Fall wird die Ausführung des nächsten Befehls oder der nächsten Teiloperation eines Blockbefehls zurückgestellt und zunächst eine Unterbrechungsroutine durchgeführt. Nach Abarbeitung dieses speziellen Unterprogramms wird das unterbrochene Programm an der Stelle der Unterbrechung fortgesetzt.

Mit Unterbrechungen lassen sich nun folgende Funktionen effizient durchführen:

- Das Auslösen bestimmter Aktionen, die durch das Eintreffen eines Unterbrechungswunsches bereits völlig gekennzeichnet sind (zum Beispiel das Rücksetzen des Computers in einen Initialzustand oder das Retten von Registerinhalten in einen nichtflüchtigen Speicher bei Versagen der Stromversorgung). Durch Verwendung von Unterbrechungen ist keine ständige Überwachung der Peripherie nötig, und die Reaktionszeit wird auf ein Minimum reduziert; das eigentliche Programm kann ohne Rücksichtnahme auf mögliche Alarmsituationen arbeiten.
- Bei Anschluß mehrerer externer Geräte, die sonst durch Polling ständig kontrolliert werden müßten, zeigt eine Unterbrechung an, daß wenigstens ein Gerät aktiv geworden ist; dies reduziert den Aufwand für das Polling auf ein Mindestmaß (Feststellung des unterbrechenden Geräts).
- Durch Zusatz-Hardware können externe Geräte sich dem Prozessor gegenüber identifizieren, sobald sie eine Unterbrechung auslösen; es ist kein Polling mehr erforderlich, jede Unterbrechung resultiert in der Ausführung eines speziell auf das unterbrechende Gerät zugeschnittenen Unterprogramms.

In allen genannten Fällen kann das eigentlich aktive Programm nahezu unabhängig von den Unterbrechungsroutinen geschrieben und betrieben werden; lediglich Programmteile, während deren Ausführung keine Unterbrechung erfolgen soll, müssen gesondert behandelt werden (sogenannte kritische Bereiche).

Unterbrechungen können priorisiert sein, das heißt, daß es Unterbrechungen gibt, deren Unterbrechungsroutinen von anderen - höher priorisierten - Unterbrechungen beeinflußt werden können, nicht aber umgekehrt. Der Z80 verfügt über zwei Priorisierungsstufen: die maskierbaren Unterbrechungen (engl. maskable interrupts) und die höher priorisierten, nicht maskierbaren Unterbrechungen (engl. non maskable interrupt).

Während die maskierbaren Unterbrechungen durch das laufende Programm gesperrt werden können, führt eine nicht maskierbare Unterbrechung stets zur Ausführung ihrer Unterbrechungsroutine.

Die maskierbaren Unterbrechungen können in drei verschiedenen Modi betrieben werden, die sich in der Form der Adressierung der Unterbrechungsroutinen durch die externen Geräte unterscheiden.

27.2 Nicht maskierbare Unterbrechungen

Der Z80 verfügt über einen Anschluß **NMI** (non maskable interrupt), auf dem der Prozessor über das Vorliegen eines Wunsches nach nicht maskierbarer Unterbrechung informiert wird; *maskieren* bedeutet in diesem Zusammenhang unterdrücken. Wird der Anschluß NMI aktiv, so führt der Z80 nach Abarbeitung des aktuellen Befehls oder der aktuellen Teileinheit eines Blockbefehls einen Unterprogrammsprung zur Adresse 0066H durch. Die Unterbrechungsroutine für die nicht maskierbaren Unterbrechungen muß damit stets an der Adresse 0066H beginnen; dort kann aber auch ein Sprung auf die eigentliche Routine stehen.

Vor Ausführung der Unterbrechungsroutine wird durch Löschen des Unterbrechungs-

Flipflops 1 – kurz IFF1 genannt – eine Reaktion auf maskierbare Unterbrechungen vorläufig unterbunden; der alte Zustand von IFF1 bleibt als Wert des Unterbrechungs-Flipflops 2 (IFF2) erhalten.

Die Unterbrechungsroutine für die nicht maskierbaren Unterbrechungen endet gewöhnlich mit einem speziellen Unterprogramm-Rücksprung-Befehl: **RETN** (return from non maskable interrupt). Dieser restauriert den alten Wert von IFF1 aus IFF2 und läßt damit wieder maskierbare Unterbrechungen zu, falls IFF2 gesetzt war; außerdem signalisiert der Befehl der externen Hardware, daß die Unterbrechungsroutine beendet ist.

Es gibt zwei typische Anwendungen für nicht maskierbare Unterbrechungen: Warmstart und Reaktion auf Stromausfall.

Die Reinitialisierung des laufenden Computers nennt man einen Warmstart (engl. warm boot). Dabei werden zumeist große Teile des Betriebssystems neu in den Speicher geladen und die Bearbeitung an einer definierten Stelle begonnen. Das Testen externer Komponenten des Computers (Speicher, Bildschirm, Tastatur) unterbleibt in der Regel beim Warmstart – im Gegensatz zum Kaltstart (engl. cold boot), der durch das Aktivieren der **RESET**-Leitung des Prozessors ausgelöst wird, normalerweise nur beim Einschalten des Geräts. Ein Warmstart wird nötig, wenn die Systemumgebung durch das laufende Programm zerstört wurde, möglicherweise bedingt durch einen Fehler.

Bei Stromausfall (engl. power fail) bleibt dem Prozessor meist eine kurze Zeitspanne, um die Inhalte von Registern und flüchtigem Speicher in nichtflüchtigen Speicher zu retten; dies erlaubt dann ein Wiederaufsetzen auf korrekte Daten nach Beheben des Defekts. Wir gehen in unserem folgenden Beispiel davon aus, daß der Hauptspeicher des Computers nichtflüchtig ist und ein kleiner Speicherbereich ab der Adresse 0085H für die Aufnahme der Registerinhalte zur Verfügung steht:

```
         ORG     0066H          ; Unterbrechungsroutine fuer
                                ; nicht maskierbare
                                ; Unterbrechungen
RETTEN:  LD      (STAPEL),SP    ; Stapel-Zeiger retten
         LD      SP,STAPEL+25   ; neuen Stapel definieren
                                ; fuer Aufnahme der
                                ; restlichen Registerinhalte
         PUSH    HL             ; HL-Register retten
         PUSH    DE             ; DE-Register retten
         PUSH    BC             ; BC-Register retten
         PUSH    AF             ; AF-Register retten
         PUSH    IY             ; IY-Register retten
         PUSH    IX             ; IX-Register retten
         EXX                    ; sekundaeren
         EX      AF,AF'         ; Registersatz holen
         PUSH    HL             ; HL'-Register retten
         PUSH    DE             ; DE'-Register retten
         PUSH    BC             ; BC'-Register retten
```

```
        PUSH    AF          ; AF'-Register retten
        LD      A,R         ; R-Register holen
        PUSH    AF          ; und retten
        INC     SP          ; Flags nicht notwendig
        LD      A,I         ; I-Register und IFF2 holen
        PUSH    AF          ; und retten
STOP:   JP      STOP        ; unterbrochenes Programm
                            ; nicht fortsetzen

; Datenbereich

STAPEL: DEFS    25          ; Speicherbereich fuer
                            ; Registerinhalte reservieren
```

Es entsteht folgende Belegung des Speicherbereichs:

0085H	SP
0087H	IFF2
0088H	I
0089H	R
008AH	AF'
008CH	BC'
008EH	DE'
0090H	HL'
0092H	IX
0094H	IY
0096H	AF
0098H	BC
009AH	DE
009CH	HL

Den Befehlszähler brauchen wir nicht sichern, da sich die Adresse der Unterbrechungsstelle auf dem Stapel befindet.

Das Wiederaufsetzen geschieht mit folgender Routine:

```
        LD      SP,STAPEL+5 ; Stapel-Zeiger fuer
                            ; Restaurieren der Register
                            ; vorbereiten
        POP     AF          ; AF'-Register restaurieren
        POP     BC          ; BC'-Register restaurieren
        POP     DE          ; DE'-Register restaurieren
        POP     HL          ; HL'-Register restaurieren
        POP     IX          ; IX-Register restaurieren
```

```
        POP     IY          ; IY-Register restaurieren
        EX      AF,AF'      ; primaeren
        EXX                 ; Registersatz holen
        POP     AF          ; AF-Register restaurieren
        POP     BC          ; BC-Register restaurieren
        POP     DE          ; DE-Register restaurieren
        POP     HL          ; HL-Register restaurieren
        PUSH    AF          ; AF-Register nochmals sichern
        LD      A,(STAPEL+4) ; Wert des R-Registers holen
        LD      R,A         ; R-Register restaurieren
        LD      SP,STAPEL+2 ; Stapel-Zeiger auf Wert von
                            ; IFF2 und I-Register richten
        POP     AF          ; Wert von IFF2 und I-Register
                            ; holen
        LD      I,A         ; I-Register restaurieren
        LD      SP,STAPEL+23 ; Stapel-Zeiger auf Wert des
                            ; AF-Registers richten
        JP      PE,UNTERB   ; IFF2 war gesetzt,
                            ; Unterbrechungen zulassen
        POP     AF          ; AF-Register restaurieren
        LD      SP,(STAPEL) ; Stapel-Zeiger restaurieren
        JP      WEITER      ; weiter an gemeinsamer
                            ; Fortsetzungsstelle
UNTERB: POP     AF          ; AF-Register restaurieren
        LD      SP,(STAPEL) ; Stapel-Zeiger restaurieren
        EI                  ; Unterbrechungen zulassen
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

27.3 Der Unterbrechungsmodus 0

Der Unterbrechungsmodus 0 ist einer der drei Modi, in denen die maskierbaren Unterbrechungen konfiguriert werden können. Beim Aktivieren der RESET-Leitung des Z80 geht dieser automatisch in den Unterbrechungsmodus 0 über, der zum Unterbrechungskonzept des Prozessors 8080 kompatibel ist. Wir können diesen Modus jederzeit gezielt durch den Befehl **IM** (interrupt mode) einstellen:

```
        IM      0           ; Unterbrechungsmodus 0
                            ; einstellen
```

Erfolgt im Modus 0 eine maskierbare Unterbrechung, ausgelöst durch Aktivieren der **INT**-Leitung des Z80, so liest der Prozessor ein Byte von dem externen Gerät ein, das die Unterbrechung bewirkt hat, interpretiert dieses Byte als Objekt-Code eines Maschinenbefehls und führt

den Befehl aus; sinnvollerweise kommt für das eingelesene Byte nur der Objekt-Code eines RST-Befehls in Frage.

Der Unterbrechungsmodus 0 gestattet damit die Unterscheidung von bis zu 8 externen Geräten, die sich über die entsprechende Adresse im RST-Befehl selbst identifizieren können; der Befehl RST 0000H ist wegen Übereinstimmung der Startadresse der Unterbrechungsroutine mit der Initialisierungsadresse beim Kaltstart aber möglicherweise nicht gut benutzbar.

An der Stelle, auf die durch den Unterprogrammaufruf RST gesprungen wird, stehen nur 8 Bytes für die jeweilige Unterbrechungsroutine zur Verfügung. Dies ist für viele Anwendungen zu knapp, weshalb an der Ansprungstelle meist nur einige Befehle der Unterbrechungsroutine stehen, gefolgt von einem Sprung auf den Rest der Unterbrechungsroutine. Beispielsweise könnte eine solche Routine mit Registerinitialisierungen beginnen:

```
ORG     0028H       ; Unterbrechungsroutine
                    ; fuer Geraet 6
PUSH    HL          ; Registerinhalt sichern
LD      HL,172FH    ; Register initialisieren
JP      0731H       ; Rest der Unterbrechungsroutine
                    ; anspringen
```

Durch Aktivieren einer maskierbaren Unterbrechung werden nachfolgende maskierbare Unterbrechungen unterdrückt. Unterbrechungsroutinen für maskierbare Unterbrechungen werden gewöhnlich durch den Befehl **RETI** (return from interrupt) abgeschlossen. Dadurch erhält das externe Gerät Kenntnis vom Abschluß der Unterbrechungsroutine.

27.4 Der Unterbrechungsmodus 1

Im Unterbrechungsmodus 0 mußte das unterbrechende Gerät den Z80 mit einem RST-Befehl versehen; dies erfordert zusätzliche Hardware. Will man ohne Zusatz-Hardware auskommen, so wählt man sich den Unterbrechungsmodus 1, bei dem stets die Adresse 0038H (wie mit einem Unterprogramm-Aufruf) angesprungen wird und keines der Geräte sich zunächst gegenüber dem Prozessor identifiziert. Das Eintreten einer Unterbrechung im Modus 1 signalisiert lediglich, daß mindestens ein Gerät bedient werden soll; sind mehrere Geräte angeschlossen, so muß der Z80 durch Polling herausfinden, welches Gerät die Unterbrechung verursacht hat.

Der Unterbrechungsmodus 1 wird durch den Befehl

```
IM      1           ; Unterbrechungsmodus 1
                    ; einstellen
```

eingestellt.

27.5 Der Unterbrechungsmodus 2

Am komfortabelsten und am besten auf das Zusammenwirken von Bausteinen aus der Z80-Familie (Zentraleinheit, Ein-/Ausgabe-Bausteine, Zeitgeber, ...) zugeschnitten ist der Unterbrechungsmodus 2; dieser wird durch den Befehl

```
IM      2       ; Unterbrechungsmodus 2
                ; einstellen
```

eingestellt. Jedem unterbrechenden Gerät wird eine eigene Unterbrechungsroutine zugeordnet. Die Anfangsadressen dieser Unterbrechungsroutinen werden in einer Tabelle zusammengefaßt. Das unterbrechende Gerät muß den korrekten Index für die Adreßtabelle liefern; dieser Index ist ein 7-Bit-Wert und wird vom Z80 auf Bit 1 bis Bit 7 des Datenbusses erwartet. Der Index wird zunächst durch Nullsetzen von Bit 0 zu einer Relativadresse gemacht; dies bedeutet, daß die Elemente der Adreßtabelle stets auf Bytes mit gerader Adresse beginnen müssen. Die Absolutadresse der benötigten Sprungadresse ergibt sich nun als Konkatenation aus dem Inhalt des Unterbrechungs-Vektor-Registers I und der berechneten Relativadresse.

Um dem I-Register die richtige Basis-Adresse zu verleihen, verwendet man den Befehl

```
LD      I,A     ; Unterbrechungs-Vektor-Register
                ; aus dem A-Register laden
```

Zuvor muß das A-Register mit der Nummer des gewünschten 256-Byte-Blocks – einer sogenannten Seite – geladen werden. Um den Inhalt des I-Registers prüfen zu können, gibt es auch den umgekehrten Befehl

```
LD      A,I     ; Inhalt des
                ; Unterbrechungs-Vektor-Registers
                ; ins A-Register kopieren
```

Wir sehen uns ein Beispiel an: Der Speicherbereich 0200H - 02FFH soll folgendermaßen belegt sein:

```
ORG     0200H   ; Tabelle der Sprungadressen fuer
                ; maskierbare Unterbrechungen im
                ; Unterbrechungsmodus 2
DEFW    2731H   ; Adresse fuer Geraet 00H
DEFW    0421H   ; Adresse fuer Geraet 01H
DEFW    16AFH   ; Adresse fuer Geraet 02H
DEFW    1932H   ; Adresse fuer Geraet 03H
:
:
:
DEFW    1256H   ; Adresse fuer Geraet 7FH
```

Weiter wollen wir annehmen, daß folgendes Programmstück ausgeführt wurde:

```
DI              ; maskierbare Unterbrechungen
                ; waehrend des Konfigurierens
                ; sperren
IM      2       ; Unterbrechungsmodus 2
                ; einstellen
LD      A,02H   ; Seitennummer laden
LD      I,A     ; Unterbrechungs-Vektor laden
EI              ; maskierbare Unterbrechungen
                ; zulassen
```

Die neuen Befehle **DI** (disable interrupts) und **EI** (enable interrupts) dienen der Maskierung beziehungsweise Demaskierung der maskierbaren Unterbrechungen. DI sperrt durch Löschen der Unterbrechungs-Flipflops IFF1 und IFF2 alle maskierbaren Unterbrechungen, bis ein EI-Befehl ausgeführt wird; nach Abarbeitung des auf den EI-Befehl folgenden Befehls sind maskierbare Unterbrechungen (durch Setzen der Unterbrechungs-Flipflops IFF1 und IFF2) zugelassen, bis ein DI-Befehl durchgeführt wird oder bis eine (maskierbare oder nicht maskierbare) Unterbrechung erfolgt ist. Das Sperren und Freigeben von maskierbaren Unterbrechungen ist in unserem Beispiel notwendig, da sonst eine während des Konfigurierens erfolgte maskierbare Unterbrechung einen falschen Modus oder einen ungültigen Unterbrechungs-Vektor antrifft.

Wenn nun ein externes Gerät eine Unterbrechung erzeugt und zum Beispiel den Wert 04H (oder 05H; beides hat dieselbe Wirkung) auf dem Datenbus liefert, so wird der Adreßtabelle die Sprungadresse 16AFH entnommen und wie mit einem Unterprogramm-Aufruf angesprungen; ab Adresse 16AFH muß dann die Unterbrechungsroutine dieses Geräts stehen.

Die Peripherie-Bausteine der Z80-Familie lassen sich so programmieren, daß sie bei der Erzeugung einer Unterbrechung den gewünschten Index auf dem Datenbus an den Prozessor liefern.

27.6 Der HALT-Befehl

Ein interessanter Befehl ist der sogenannte Halt-Befehl **HALT** (halt). Nach Ausführung eines HALT-Befehls geht der Z80 in einen Wartezustand über, in dem solange nur NOP-Befehle ausgeführt werden (ohne Veränderung des Befehlszählers), bis eine Unterbrechung erfolgt. Dies ist nützlich, wenn das Programm Eingabedaten erwartet, die von einem beliebigen von mehreren externen Geräten kommen können. Sobald ein Gerät Daten bereithält, löst es eine Unterbrechung aus, die in einem Aufruf der jeweiligen Unterbrechungsroutine resultiert; dort wird dann für die Abholung der Daten und anschließende Verarbeitung gesorgt. Der HALT-Befehl wird stets dort verwendet, wo ohne Anforderung von einem externen Gerät keine weiteren Aktionen sinnvoll sind, zum Beispiel:

```
        :
        :                           ; System initialisieren
        :
        HALT                        ; auf Eingaben warten
```

27.7 Unterbrechungen und Pufferbearbeitung

Wird ein Puffer durch ein externes Gerät gefüllt und durch ein ständig aktives Programm geleert (oder umgekehrt), so muß verhindert werden, daß eine Unterbrechung durch den Produzenten gerade dann erfolgt, wenn der Puffer sich durch Aktionen des Konsumenten in einem inkonsistenten Zustand befindet.

Wir sperren für das Unterprogramm zum Leeren des Puffers die maskierbaren Unterbrechungen; das besagte Programmstück heißt *kritischer Bereich.*

Am Beispiel der Pufferbearbeitungs-Programme demonstrieren wir die dazu benutzte Technik:

```
LEERE:  DI                          ; maskierbare Unterbrechungen
                                    ; sperren
        CALL    LEER                ; pruefen, ob Puffer leer
        JP      Z,LENDE             ; Puffer leer, Fehler
        PUSH    HL                  ; Register-
        PUSH    DE                  ; inhalte sichern
        LD      HL,(KZEIG)          ; Konsumenten-Zeiger holen
        LD      A,(HL)              ; Zeichen aus Puffer nehmen
        PUSH    AF                  ; und sichern
        INC     HL                  ; auf naechstes Zeichen zeigen
        LD      (KZEIG),HL          ; Konsumenten-Zeiger sichern
        LD      DE,(PZEIG)          ; Produzenten-Zeiger holen
        OR      A                   ; pruefen, ob Puffer-Zeiger
        SBC     HL,DE               ; uebereinstimmen
        CALL    Z,INIT              ; Puffer leer,
                                    ; Puffer-Zeiger ruecksetzen
        POP     AF                  ; alle
        POP     DE                  ; Register
        POP     HL                  ; restaurieren
LENDE:  EI                          ; maskierbare Unterbrechungen
                                    ; wieder freigeben
        RET
```

Vorsicht! Vergessene EI-Befehle führen oft zu Sackgassen, aus denen sich das Programm nicht mehr befreien kann; Warm- oder sogar Kaltstart sind dann meist die einzigen Möglichkeiten, die Kontrolle über den Prozessor wieder zu erlangen.

Übungen

1. In dem Unterprogramm LEERE befinden sich Programmstücke, die eigentlich nicht zum kritischen Bereich gehören. Modifiziere das Unterprogramm so, daß nur für die Programmstücke die maskierbaren Unterbrechungen gesperrt werden, für die das unumgänglich ist.

2. Wie hängen eintrittsinvariante Unterprogramme und Unterbrechungen zusammen?

27.8 Schnelle Unterbrechungsroutinen

In vielen Anwendungen ist es wichtig, auf Unterbrechungen innerhalb einer möglichst kurzen Zeit zu reagieren und die Unterbrechungsbehandlung selbst ebenfalls möglichst schnell abzuschließen. Beim Betreten einer Unterbrechungsroutine weiß man natürlich nicht, welche Register des unterbrochenen Programms gerade gültige Werte besitzen; das Sichern aller durch die Unterbrechungsroutine benutzter Register ist deswegen meist angebracht.

Leider benötigen Ladebefehle, in denen explizite Speicheradressen vorkommen, und Stapelbefehle relativ viel Zeit. Man verwendet deshalb meist mit Vorteil den sekundären Registersatz für die Unterbrechungsroutine und den primären Registersatz für das ständig laufende Programm. Das Tauschen der Registersätze selbst benötigt wenig Zeit. Wir schreiben das Unterprogramm zum Füllen eines Puffers auf diese Weise um:

```
FUELLE:  EXX                    ; sekundaeren
         EX     AF,AF'          ; Registersatz holen
         CALL   VOLL            ; pruefen, ob Puffer voll
         JP     Z,FENDE         ; Puffer voll, Fehler
         LD     HL,(PZEIG)      ; Produzenten-Zeiger holen
         LD     (HL),A          ; Zeichen ablegen
         INC    HL              ; auf naechstes Zeichen zeigen
         LD     (PZEIG),HL      ; Produzenten-Zeiger sichern
FENDE:   EXX                    ; primaeren
         EX     AF,AF'          ; Registersatz holen
         RET
```

28
Verschiebbare Programme

Ein Programm heißt *verschiebbar* oder auch *relocierbar* (engl. relocatable), wenn es nach dem Assemblieren und Binden an jede beliebige Stelle des Speichers gebracht werden kann und dort uneingeschränkt lauffähig ist. Das Verschieben des Programms erfolgt so, daß jedes Byte des Objekt-Codes um dieselbe Relativadresse in dieselbe Richtung bewegt wird. Verschiebbare Unterprogramme sind wichtig, wenn eine Bibliothek von Unterprogrammen benötigt wird, aus denen ohne Assemblierungsvorgang ein komplettes Programm zusammengestellt werden kann.

28.1 Relative Sprünge und Unterprogramm-Rücksprünge

Wenn ein Programm einen direkten absoluten Sprung enthält, dessen Ziel in diesem Programm liegt (es sind ja auch Sprünge in Betriebssystem-Routinen möglich, die nicht zum Programm selbst gehören), so ist dieses Programm mit Sicherheit nicht verschiebbar; nach dem Verschieben des Programms würde der Sprung ja immer noch zur selben Adresse führen, an der aber das gewünschte Programmstück gar nicht mehr steht. Dasselbe Argument trifft auch für Unterprogramm-Aufrufe durch CALL- oder RST-Befehle zu.

Relative Sprünge dagegen enthalten nicht die anzuspringende Adresse, sondern die Sprungdistanz; diese ändert sich aber durch das Verschieben gerade nicht. Unterprogramm-Rücksprünge enthalten auch keine Adressen, sondern sorgen durch Beschaffen der Rückkehradresse vom Stapel für einen korrekten Rücksprung; auch die Unterprogramm-Rücksprünge sind daher zur Verwendung in verschiebbaren Programmen geeignet.

Wir sehen also: Programme oder Programmstücke, die ausschließlich mit relativen Sprüngen (JR-Befehle und DJNZ-Befehl) und Unterprogramm-Rücksprüngen (RET-Befehle, RETI-Befehl, RETN-Befehl) arbeiten, sind verschiebbar, wenn sie keine Befehle enthalten, in denen die Adressen von Daten vorkommen, welche mit dem Programm verschoben werden sollen.

Hier ein Beispiel eines verschiebbaren Unterprogramms: Eine Modifikation des Unterprogramms HEXASC aus Unterkapitel 19.1, das die ASCII-Codierung einer Hex-Ziffer berechnet.

```
HEXASC:  CP      10          ; A-Register auf
                             ; Dezimalziffer testen
         JR      C,DEZIMA    ; Dezimalziffer im A-Register
         ADD     A,'A'-0AH-'0' ; Korrektur fuer Buchstabe
DEZIMA:  ADD     A,'0'       ; ASCII-Darstellung berechnen
         RET                 ; Ruecksprung ins Hauptprogramm
```

Das Unterprogramm enthält weder absolute Sprünge noch Unterprogramm-Aufrufe; alle benötigten Daten stehen in Registern oder sind im Objekt-Code des Unterprogramms enthalten. Das Unterprogramm ist deshalb verschiebbar.

Wie man Daten zu adressieren hat, wenn man verschiebbare Programme erhalten möchte, lernen wir im Unterkapitel 28.4.

Übungen

1. Überlegen Sie sich die Einschränkungen, die nach dem bisher Gesagten für verschiebbare Programme resultieren.

28.2 Das Beschaffen der Basis-Adresse

Wie wir bereits wissen, kann man Code und Daten indirekt über Register adressieren: bei indirekten Sprüngen mittels der Indexregister und des Registers HL, bei indirekter Datenadressierung mittels der Indexregister und der Register BC, DE und (vor allem) HL. Um indirekte Adressierung durchführen zu können, müssen wir allerdings über die Adressen verfügen, um sie in die entsprechenden Code- oder Daten-Adreßregister bringen zu können. Wir kennen auf jeden Fall die Relativadressen bezüglich des Programmanfangs, da diese durch das Verschieben eines Programms nicht verändert werden. Was wir noch brauchen, ist die absolute Adresse des Programmanfangs.

Eine schöne Möglichkeit, sich diese Basis-Adresse in einem Register zu verschaffen, ist folgende: Wir schreiben uns ein Unterprogramm, das nicht mit dem Programm verschoben wird, sondern stets an einer festen Stelle des Speichers bleibt. Dieses Unterprogramm rufen wir aus unserem verschobenen Programm auf. Dadurch wird die Rückkehradresse, also die Adresse des nächsten auf den CALL-Befehl folgenden Befehls, auf den Stapel gebracht; diese Rückkehradresse wird nun unsere Basis-Adresse. Das Unterprogramm holt die Rückkehradresse in ein Indexregister oder das HL-Register und führt einen indirekten Sprung damit aus, der damit wie ein RET-Befehl wirkt. Nach Rückkehr aus dem Unterprogramm befindet sich dann die Basis-Adresse im entsprechenden Register.

Die drei Unterprogramme für die Register IX, IY und HL lauten:

```
XBASIS:   POP    IX      ; Basis-Adresse ins IX-Register
                         ; bringen
          JP     (IX)    ; Ruecksprung zur Basis-Adresse

YBASIS:   POP    IY      ; Basis-Adresse ins IY-Register
                         ; bringen
          JP     (IY)    ; Ruecksprung zur Basis-Adresse

HBASIS:   POP    HL      ; Basis-Adresse ins HL-Register
                         ; bringen
          JP     (HL)    ; Ruecksprung zur Basis-Adresse
```

Die drei Unterprogramme sind zwar selbst verschiebbar; dies können wir aber meist nicht ausnutzen, da sonst die aufrufenden Programme nicht wissen, wo die Unterprogramme stehen.

Übungen

1. Schreibe entsprechende Unterprogramme, welche die Basis-Adresse im BC-Register beziehungsweise DE-Register liefern, sonst aber keine Register benutzen.

28.3 Indirekte Sprünge

Wir wollen uns nun überlegen, wie wir die Basis-Adresse benutzen können, um absolute Sprünge in einem verschiebbaren Programm auszuführen. Betrachte dazu folgendes Beispiel eines nicht verschiebbaren Programmstücks:

```
          :
          :
          :
          JP     ZIEL     ; absolute Adresse anspringen
          :
          :
          :
ZIEL:                     ; Sprungziel
          :
          :
          :
```

Wir können nun das Programmstück ganz schematisch modifizieren und dadurch verschiebbar machen:

```
          :
          :
          :
          CALL      HBASIS          ; Basis-Adresse BASIS
                                    ; im HL-Register besorgen
BASIS:    LD        DE,ZIEL-BASIS   ; Relativ-Adresse
                                    ; ins DE-Register bringen
          ADD       HL,DE           ; Ziel-Adresse ZIEL berechnen
          JP        (HL)            ; Ziel-Adresse ZIEL anspringen
          :
          :
          :
ZIEL:                               ; Ziel-Adresse
          :
          :
          :
```

Die Umformung hat die gewünschte Wirkung, da die Relativ-Adresse ZIEL—BASIS durch das Verschieben nicht geändert wird.

Das Vorgehen im Falle der Verwendung eines Indexregisters ist ganz analog.

Übungen

1. Schreibe ein Unterprogramm, dem im DE-Register eine Relativ-Adresse übergeben wird, und das selbständig den gewünschten indirekten Sprung ausführt (wie im letzten Beispiel).

2. Versuche folgendes nicht verschiebbare Programmstück schematisch in ein verschiebbares Programmstück umzuformen:

```
          :
          :
          :
          CALL      UP              ; Unterprogramm aufrufen
RUECK:                              ; Rueckkehr-Adresse
          :
          :
          :
```

```
UP:                                    ; Unterprogramm
        :
        :
        :
        RET                            ; Ruecksprung zur Adresse RUECK
```

28.4 Indirekte Daten-Adressierung

Für indirekte Daten-Adressierung in verschiebbaren Programmen gibt es zwei Möglichkeiten: Zunächst kann, wie im vorhergehenden Unterkapitel gezeigt, die absolute Adresse der zu bearbeitenden Speicherzelle ins HL-Register gebracht werden; das HL-Register wird ab da in gewohnter Weise als Datenadreßregister benutzt, also beispielsweise:

```
        :
        :
        :
        CALL    HBASIS          ; Basis-Adresse BASIS
                                ; im HL-Register beschaffen
BASIS:  LD      DE,DATEN-BASIS  ; Relativ-Adresse
                                ; ins DE-Register bringen
        ADD     HL,DE           ; Zeiger auf DATEN berechnen
        LD      A,(HL)          ; Daten bearbeiten
        :
        :
        :
DATEN:  DEFB    '*'             ; Anfang des Daten-Bereichs
        :
        :
        :
```

Liegen die Daten in der Nähe der Basis-Adresse, so kann als zweite Möglichkeit indirekte Daten-Adressierung durch Indexregister gewählt werden. Betrachte dazu folgendes Beispiel:

```
        :
        :
        :
        CALL    XBASIS              ; Basis-Adresse BASIS
                                    ; im IX-Register beschaffen
BASIS:  LD      A,(IX+DATEN-BASIS)  ; Daten bearbeiten
        :
        :
        :
```

```
DATEN:  DEFB    '*'             ; Anfang des Daten-Bereichs
        :
        :
        :
```

Liegen die Daten zu weit von der Basis-Adresse entfernt (die Relativ-Adressen in Indexregister-Befehlen sind ja stets im Bereich −128 bis +127), so kombinieren wir beide Methoden; zuerst holen wir die Basis-Adresse, berechnen aus dieser dann eine für die Daten geeignete neue Basis-Adresse und adressieren von dieser ausgehend mittels eines Indexregisters. Dazu ebenfalls ein Beispiel:

```
        :
        :
        :
        CALL    XBASIS          ; Basis-Adresse BASIS
                                ; im IX-Register beschaffen
BASIS:  LD      DE,DATEN-BASIS  ; Relativ-Adresse zu neuer Basis
                                ; DATEN ins DE-Register bringen
        ADD     IX,DE           ; neue Basis-Adresse DATEN
                                ; im IX-Register berechnen
        LD      A,(IX+00H)      ; Daten bearbeiten
        :
        :
        :
DATEN:                          ; neue Basis-Adresse,
                                ; Anfang des Datenbereichs
        DEFB    '*'
        :
        :
        :
```

Die neue Basis-Adresse braucht nicht unbedingt am Anfang des Datenbereichs liegen; wird sie genau in die Mitte des Datenbereichs gelegt, so läßt sich die maximale Anzahl von 256 Bytes Datenspeicher adressieren.

29
Anspruchsvolle Programmbeispiele

In diesem abschließenden Kapitel sollen noch einige interessante Programmierprobleme besprochen werden, die gelernte Methoden demonstrieren und vertiefen. All diese Probleme besitzen eine gewisse Bedeutung für praktische Anwendungen, auch wenn sie in der vorliegenden Form vielleicht nicht direkt in ein Projekt eingebaut werden können; Abstraktionen von konkreten Vorgaben sind in einem solchen Buch nun einmal nicht zu vermeiden. Das Verständnis der Lösungen wird Dir bei ähnlichen Anwendungen sicherlich eine Hilfe sein.

29.1 Zufalls-Zahlen-Generator

Für viele Anwendungen in Statistik und Simulation sowie beim Einsatz heuristischer Methoden benötigt man Zahlen, die zufällig einer bestimmten Zahlenmenge entnommen sind. Wir wollen uns deshalb einen Zufalls-Zahlen-Generator schreiben.

Der Z80 verfügt über ein Register, das wir bis jetzt noch nicht besprochen haben: das Speicher-Auffrisch-Register R. Dieses Register benötigt der Prozessor, um den flüchtigen Speicher in bestimmten Abständen wieder aufzufrischen, so daß einem Verlust der Information des Speichers vorgebeugt wird. Wir können das R-Register aber auch benutzen, um uns eine Zufalls-Zahl im Bereich der ganzen Zahlen zwischen 0 und 127 zu verschaffen.

Das R-Register erhält beim Rücksetzen des Prozessors durch die RESET-Leitung den Wert 00H. Jedesmal nach dem Holen eines Befehls werden die niederwertigen 7 Bits des R-Registers inkrementiert; das höchstwertige Bit behält seinen Wert. Durch den Befehl

```
LD          R,A         ; Inhalt des A-Registers
                        ; ins R-Register bringen
```

kann das R-Register zu Testzwecken geladen werden; dabei kann auch das höchstwertige Bit des R-Registers neu gesetzt werden.

Wenn wir ein Programm aus der Betriebssystemebene heraus starten, so ist zwischen dem Rücksetzen des Prozessors und dem Starten des Programms eine im allgemeinen zufällige Zeitspanne vergangen; dies bedeutet, daß der Prozessor in der Zwischenzeit eine zufällige Anzahl von Befehlen ausgeführt hat. Das R-Register trägt deshalb einen (bis auf das höchstwertige Bit) zufälligen Wert, den wir uns mit dem Befehl

```
        LD          A,R         ; Inhalt des R-Registers
                                ; ins A-Register bringen
```

verschaffen können.

Soll eine weitere Zufalls-Zahl nach derselben Methode im gleichen Programm gewonnen werden, so muß dafür gesorgt werden, daß zwischen den beiden Befehlen zum Beschaffen der Zufalls-Zahlen wieder eine zufällige Zeitspanne vergeht. Dies kann zum Beispiel durch Warten auf eine Benutzereingabe oder auf ein externes Signal (Unterbrechung) geschehen.

Man kann als Alternative dazu auch aus der ersten (echten) Zufallszahl eine Folge von Zahlen berechnen, die annähernd zufällig verteilt sind. Diese Zahlen nennt man *Pseudo-Zufalls-Zahlen.*

Eine einfache Methode zur Erzeugung von Pseudo-Zufalls-Zahlen ist folgende: zu einer vorgegebenen Pseudo-Zufalls-Zahl z wird eine neue Pseudo-Zufalls-Zahl z' durch $z' = ((z * r) + s) \bmod 128$ berechnet. Dabei sind r und s positive ganze Zahlen, die kleiner als 128 sind; r und s müssen außerdem so gewählt sein, daß die Folge der berechneten Zahlen wirklich alle ganzen Zahlen zwischen 0 und 127 enthält.

Folgendes Unterprogramm bildet eine neue Pseudo-Zufalls-Zahl, welche nach dem beschriebenen Verfahren mit den Parametern r=5 und s=3 berechnet wird (die Zufalls-Zahlen stehen im A-Register):

```
PSEUDO:  LD      C,A        ; Pseudo-Zufalls-Zahl sichern
         ADD     A,A        ; Zahl
         ADD     A,A        ; mit 5
         ADD     A,C        ; multiplizieren
         ADD     A,3        ; neue Pseudo-Zufalls-Zahl bilden
         AND     7FH        ; oberstes Bit ausmaskieren
         RET
```

29.2 Bildschirmsteuerung

Dieses Unterkapitel setzt Aufgabe 3 aus Unterkapitel 9.4 fort. Dort war eine durch numerische Eingabe codierte Cursor-Bewegung fiktiv durchzuführen. Neben dieser gedachten Bewegung wollen wir für eine bestimmte Art von Bildschirm die Bewegung jetzt auch tatsächlich ausführen lassen. Es handelt sich dabei um den Bildschirm nach der ANSI-Norm, die mittlerweile sehr stark in Gebrauch ist. Nach dieser Norm werden *Escape-Sequenzen* (siehe Kapitel »Zeichen«) zur Steuerung der Cursor-Bewegung verwendet, und zwar

Escape-Sequenz	Funktion
ESC [A	eine Zeile aufwärts
ESC [B	eine Zeile abwärts
ESC [C	ein Zeichen nach rechts
ESC [D	ein Zeichen nach links

Wir nehmen nun wieder an, daß die gewünschte Bewegung durch eine ASCII-Ziffer codiert ist, und zwar nach folgendem Schema:

1 abwärts und links
2 abwärts
3 abwärts und rechts
4 links
5 linke obere Ecke
6 rechts
7 aufwärts und links
8 aufwärts
9 aufwärts und rechts

Dabei sollten Sie sich folgendes Tastenfeld vorstellen:

```
7   8   9
4   5   6
1   2   3
```

Bei der Auswertung der Eingabe behandeln wir zunächst vier Fälle:

A abwärts (möglicherweise mit links oder rechts): 1, 2, 3
B aufwärts (möglicherweise mit links oder rechts): 7, 8, 9
C linke obere Ecke: 5
D alle übrigen Fälle: 4, 6

Die entsprechende Bewegung des Cursors führen wir auf den Koordinatenregistern und auf dem Bildschirm selbst aus.

Anschließend bilden wir die Fälle B und D durch Subtraktion des Werts 6 beziehungsweise 3 auf den Fall A ab. Es sind nun für die seitliche Bewegung drei Fälle zu behandeln:

a links: 1
b keine Bewegung: 2
c rechts: 3

Wir führen auch diese Bewegung auf den Koordinaten und auf dem Bildschirm aus. In den Fällen A, B, a, b achten wir jeweils darauf, daß der zulässige Bildschirmbereich nicht verlassen

wird; dieser Bereich wird durch die kleinste und die größte Zeilennummer sowie die kleinste und größte Spaltennummer gegeben und kann auch ein rechteckiger Ausschnitt des vollen Bildschirms sein (ein sogenanntes *Fenster*).

Die Escape-Sequenzen legen wir als Folgen von jeweils 3 Bytes im Speicher ab. Zur Ausgabe dient eine spezielle Routine SEQUNZ, die im HL-Register mit einem Zeiger auf die Sequenz versorgt wird; die eigentliche Weitergabe der Zeichen an den Bildschirm überlassen wir einer Routine AUSGAB, die nicht näher spezifiziert wird (Hardware-abhängig).

```
SEQUNZ:  LD      B,3       ; Laenge der Sequenz
ZEICH:   LD      A,(HL)    ; Zeichen holen
         CALL    AUSGAB    ; Zeichen ausgeben
         INC     HL        ; auf naechstes Zeichen zeigen
         DJNZ    ZEICH     ; gesamte Sequenz ausgeben
         RET
```

Die Sequenzen selbst lauten:

```
ESC      EQU     1BH       ; Escape-Zeichen
AUF:     DEFB    ESC       ; Sequenz
         DEFB    '['       ; fuer
         DEFB    'A'       ; aufwaerts
AB:      DEFB    ESC       ; Sequenz
         DEFB    '['       ; fuer
         DEFB    'B'       ; abwaerts
RECHTS:  DEFB    ESC       ; Sequenz
         DEFB    '['       ; fuer
         DEFB    'C'       ; rechts
LINKS:   DEFB    ESC       ; Sequenz
         DEFB    '['       ; fuer
         DEFB    'D'       ; links
```

Die Registerbelegung für unser Problem soll sein:

Parameter:

A von der Tastatur gelesener Code
B vertikale Koordinate (nicht negativ)
C horizontale Koordinate (nicht negativ)
D größte Zeilennummer
E kleinste Zeilennummer
H größte Spaltennummer
L kleinste Spaltennummer

Hilfsregister:

A' auszugebendes Zeichen
B' Zähler
HL' Zeiger auf Escape-Sequenz

Die Decodierungsroutine lautet damit:

```
DECODE:  CP    4          ; auf abwaerts testen
         JP    NC,NABW    ; nicht abwaerts
         CALL  ABW        ; abwaerts
         JP    SEITE      ; seitliche Bewegung
NABW:    SUB   3          ; Vertikal-Korrektur
         CP    4          ; auf aufwaerts testen
         JP    C,NAUFW    ; nicht aufwaerts
         CALL  AUFW       ; aufwaerts
         SUB   3          ; Vertikal-Korrektur
         JP    SEITE      ; seitliche Bewegung
NAUFW:   CP    2          ; auf linke obere Ecke testen
         JP    NZ,SEITE   ; nicht linke obere Ecke
         CALL  ECKE       ; linke obere Ecke
         JP    FERTIG     ; Ende der Ausgabe
SEITE:   CP    1          ; auf links testen
         JP    NZ,NLI     ; nicht links
         CALL  LIN        ; links
         JP    FERTIG     ; Ende der Ausgabe
NLI:     CP    2          ; auf Mitte testen
         JP    Z,FERTIG   ; Ende der Ausgabe
         CALL  RECH       ; rechts
FERTIG:  RET
```

Wir optimieren das Programm noch etwas:

```
DECODE:  CP    4          ; auf abwaerts testen
         JP    NC,NABW    ; nicht abwaerts
         CALL  ABW        ; abwaerts
         JP    SEITE      ; seitliche Bewegung
NABW:    SUB   3          ; Vertikal-Korrektur
         CP    4          ; auf aufwaerts testen
         JP    C,NAUFW    ; nicht aufwaerts
         CALL  AUFW       ; aufwaerts
         SUB   3          ; Vertikal-Korrektur
         JP    SEITE      ; seitliche Bewegung
```

```
NAUFW:  CP      2       ; auf linke obere Ecke testen
        JP      Z,ECKE  ; linke obere Ecke,
                        ; Ende der Ausgabe
SEITE:  DEC     A       ; auf links testen
        JP      Z,LIN   ; links,
                        ; Ende der Ausgabe
        DEC     A       ; auf Mitte testen
        RET     Z       ; Ende der Ausgabe
        JP      RECH    ; rechts,
                        ; Ende der Ausgabe
```

Die Routinen für die einzelnen Bewegungen lauten:

```
ABW:    EX      AF,AF'  ; Code sichern
        LD      A,B     ; Zeilennummer holen
        CP      D       ; mit groesster Zeilennummer
                        ; vergleichen
        JP      Z,ENDEAB ; Cursor am unteren Rand
        INC     B       ; neue Zeilennummer eintragen
        EXX             ; sekundaeren Registersatz holen
        LD      HL,AB   ; Zeiger auf Escape-Sequenz
        CALL    SEQUNZ  ; Sequenz ausgeben
        EXX             ; primaeren Registersatz holen
ENDEAB: EX      AF,AF'  ; Code wieder beschaffen
        RET
AUFW:   EX      AF,AF'  ; Code sichern
        LD      A,B     ; Zeilennummer holen
        CP      E       ; mit kleinster Zeilennummer
                        ; vergleichen
        JP      Z,ENDAUF ; Cursor am oberen Rand
        DEC     B       ; neue Zeilennummer eintragen
        EXX             ; sekundaeren Registersatz holen
        LD      HL,AUF  ; Zeiger auf Escape-Sequenz
        CALL    SEQUNZ  ; Sequenz ausgeben
        EXX             ; primaeren Registersatz holen
ENDAUF: EX      AF,AF'  ; Code wieder beschaffen
        RET
RECH:   EX      AF,AF'  ; Code sichern
        LD      A,C     ; Spaltennummer holen
        CP      H       ; mit groesster Spaltennummer
                        ; vergleichen
        JP      Z,ENDERE ; Cursor am rechten Rand
        INC     C       ; neue Spaltennummer eintragen
```

```
        EXX                         ; sekundaeren Registersatz holen
        LD      HL,RECHTS           ; Zeiger auf Escape-Sequenz
        CALL    SEQUNZ              ; Sequenz ausgeben
        EXX                         ; primaeren Registersatz holen
ENDERE: EX      AF,AF'              ; Code wieder beschaffen
        RET
LIN:    EX      AF,AF'              ; Code sichern
        LD      A,C                 ; Spaltennummer holen
        CP      L                   ; mit kleinster Spaltennummer
                                    ; vergleichen
        JP      Z,ENDELI            ; Cursor am linken Rand
        DEC     C                   ; neue Spaltennummer eintragen
        EXX                         ; sekundaeren Registersatz holen
        LD      HL,LINKS            ; Zeiger auf Escape-Sequenz
        CALL    SEQUNZ              ; Sequenz ausgeben
        EXX                         ; primaeren Registersatz holen
ENDELI: EX      AF,AF'              ; Code wieder beschaffen
        RET
ECKE:   EX      AF,AF'              ; Code sichern
        LD      A,B                 ; Zeilennummer laden
        SUB     E                   ; Abstand von kleinster
                                    ; Zeile berechnen
        JP      Z,KEIAUF            ; Cursor in oberster Zeile
        EXX                         ; sekundaeren Registersatz holen
        LD      B,A                 ; Anzahl der Aufwaerts-Bewegungen
                                    ; in Zaehler laden
ECKAUF: PUSH    BC                  ; Registerinhalt sichern
        LD      HL,AUF              ; Zeiger auf Escape-Sequenz laden
        CALL    SEQUNZ              ; Escape-Sequenz ausgeben
        POP     BC                  ; Register restaurieren
        DJNZ    ECKAUF              ; zum oberen Rand gehen
        EXX                         ; primaeren Registersatz holen
KEIAUF: LD      A,C                 ; Spaltennummer laden
        SUB     L                   ; Abstand von kleinster
                                    ; Spalte berechnen
        JP      Z,KEINLI            ; Cursor am linken Rand
        EXX                         ; sekundaeren Registersatz holen
        LD      B,A                 ; Anzahl der Links-Bewegungen
                                    ; in Zaehler laden
ECKLI:  PUSH    BC                  ; Registerinhalt sichern
        LD      HL,LINKS            ; Zeiger auf Escape-Sequenz laden
        CALL    SEQUNZ              ; Escape-Sequenz ausgeben
        POP     BC                  ; Register restaurieren
```

```
        DJNZ    ECKLI       ; zum linken Rand gehen
        EXX                 ; primaeren Registersatz holen
KEINLI: LD      B,E         ; Koordinaten
        LD      C,L         ; entsprechend setzen
        EX      AF,AF'      ; Code wieder holen
        RET
```

29.3 Fehler-korrigierende Codes

Bei der Übertragung von Information über Datenleitungen können Störungen auftreten, die zur Verfälschung der Daten führen. Man kann sich dagegen in gewissem Maße schützen, indem man zusammen mit den Daten Prüfinformation überträgt. Aus der Prüfinformation ist nach der Übertragung ersichtlich, ob Fehler aufgetreten und wo diese lokalisiert sind; Voraussetzung ist, daß nicht allzu viele Fehler in einem kurzen Abschnitt übertragener Daten vorgefallen sind.

Eine sehr systematische und deshalb gern verwendete Codierung ist die Hamming-Codierung. Wir fassen dazu je 4 Bits Information zu einer Einheit zusammen und fügen diesen in ganz gezielter Form 4 Bits Prüfinformation bei. Die resultierenden 8 Bits werden übertragen. Wird von diesen 8 Bits nur ein einziges gestört, so kann die ursprüngliche Information durch Decodierung zurückgewonnen werden. Werden 2 der 8 Bits gestört, so kann der Decodierer zwar keine Korrektur mehr vornehmen, jedoch die Verfälschung immerhin noch erkennen. Bei mehr als 2 aufgetretenen Fehlern kann der Decodierer im allgemeinen keine Aussage mehr machen.

Wir nehmen im folgenden stets an, daß höchstens 2 Fehler bei der Übertragung auftreten. Die übertragenen 8 Bits setzen sich folgendermaßen zusammen:

b_0	Informationsbit x_0
b_1	Informationsbit x_1
b_2	Informationsbit x_2
b_3	Informationsbit x_3
b_4	Prüfbit $y_0 = x_0$ **xor** x_1 **xor** x_2
b_5	Prüfbit $y_1 = x_0$ **xor** x_1 **xor** x_3
b_6	Prüfbit $y_2 = x_1$ **xor** x_2 **xor** x_3
b_7	Prüfbit $y_3 = x_0$ **xor** x_2 **xor** x_3

Bei der Decodierung werden zunächst 4 Prüfsummen ausgewertet:

$s_0 = b_0$ **xor** b_1 **xor** b_2 **xor** b_4
$s_1 = b_0$ **xor** b_1 **xor** b_3 **xor** b_5
$s_2 = b_1$ **xor** b_2 **xor** b_3 **xor** b_6
$s_3 = b_0$ **xor** b_2 **xor** b_3 **xor** b_7

Die Fehlerkorrektur beziehungsweise Fehlererkennung erfolgt dann mittels folgender Tabelle:

s_0	s_1	s_2	s_3	x_0	x_1	x_2	x_3
0	0	0	0	b_0	b_1	b_2	b_3
1	0	0	0	b_0	b_1	b_2	b_3
0	1	0	0	b_0	b_1	b_2	b_3
1	1	0	0	Fehler nicht korrigierbar			
0	0	1	0	b_0	b_1	b_2	b_3
1	0	1	0	Fehler nicht korrigierbar			
0	1	1	0	Fehler nicht korrigierbar			
1	1	1	0	b_0	$1-b_1$	b_2	b_3
0	0	0	1	b_0	b_1	b_2	b_3
1	0	1	1	Fehler nicht korrigierbar			
0	1	0	1	Fehler nicht korrigierbar			
1	1	0	1	$1-b_0$	b_1	b_2	b_3
0	0	1	1	Fehler nicht korrigierbar			
1	0	1	1	b_0	b_1	$1-b_2$	b_3
0	1	1	1	b_0	b_1	b_2	$1-b_3$
1	1	1	1	Fehler nicht korrigierbar			

Wir programmieren jetzt dieses Verfahren. Für jedes der vier Informationsbits legen wir eine Maske an, in der genau dort eine Eins steht, wo das entsprechende Bit in der Codierung auftaucht:

Informationsbit	Maske
x_0	10110001
x_1	01110010
x_2	11010100
x_3	11101000

Jede dieser Masken wird genau dann mittels XOR in die Codierung einbezogen, wenn das entsprechende Informationsbit den Wert Eins trägt; ansonsten wird die Maske nicht berücksichtigt. Die Routine zur Codierung lautet damit (die vier Informationsbits werden im niederwertigen Nibble des C-Registers angeliefert, die auszusendenden acht Bit stehen anschließend im A-Register):

```
; Unterprogramm fuer Hamming-Codierung

HAMM:   XOR     A       ; Akkumulator loeschen
        LD      B,4     ; Anzahl der Informationsbits
        LD      HL,CMASK ; Zeiger auf die Codier-Masken
CODE:   RRC     C       ; Informationsbit besorgen
        JP      NC,NULL ; Informationsbit ist Null,
                        ; Maske wird ignoriert
        XOR     (HL)    ; Maske in Codierung einbeziehen
NULL:   INC     HL      ; auf naechste Maske zeigen
        DJNZ    CODE    ; alle Informationsbits
                        ; verarbeiten
        RET

; Masken fuer Hamming-Codierung

CMASK:  DEFB    10110001B ; fuer Informationsbit 0
        DEFB    01110010B ; fuer Informationsbit 1
        DEFB    11010100B ; fuer Informationsbit 2
        DEFB    11101000B ; fuer Informationsbit 3
```

Bei der Decodierung gehen wir den umgekehrten Weg. Wir berechnen die Prüfsummen s_0 bis s_3 – diese werden auch *Syndrome* genannt –, indem wir für jedes der empfangenen acht Bits eine Maske mittels XOR in die Decodierung einbeziehen, falls das entsprechende Bit gesetzt ist. Die Masken lauten hier:

Datenbit	Maske
b_0	1011
b_1	0111
b_2	1101
b_3	1110
b_4	0001
b_5	0010
b_6	0100
b_7	1000

Das Syndrom verwenden wir als Index in einer Tabelle, in der wiederum Masken zur Korrektur des fehlerhaften Bits stehen (falls überhaupt ein Fehler aufgetreten ist, der ein Informationsbit verfälscht hat). Diese Masken haben als Bit 7 den Wert Eins, falls der Fehler nicht korrigierbar ist. Die Korrekturmasken lauten also:

Syndrom	Korrekturmaske
0	00000000
1	00000000
2	00000000
3	10000000
4	00000000
5	10000000
6	10000000
7	00000010
8	00000000
9	10000000
10	10000000
11	00000001
12	10000000
13	00000100
14	00001000
15	10000000

Die folgende Routine erhält im C-Register die acht übertragenen Datenbits. Sie generiert daraus im A-Register einen Nibble, der die korrigierten Informationsbits enthält, falls höchstens ein Fehler aufgetreten ist. Falls eine Fehlerkorrektur nicht möglich war, ist das Bit 7 des A-Registers gesetzt.

```
; Unterprogramm zur Syndrom-Decodierung des Hamming-Codes

DECODE:   XOR    A          ; Akkumulator loeschen
          LD     B,8        ; Anzahl der Datenbits
          LD     HL,SMASK   ; Zeiger auf Masken zur
                            ; Berechnung der Syndrome
SYNDRO:   RRC    C          ; Datenbit besorgen
          JP     NC,NICHTS  ; Maske wird nicht einbezogen
          XOR    (HL)       ; Maske verknuepfen
NICHTS:   INC    HL         ; auf naechste Maske zeigen
          DJNZ   SYNDRO     ; alle Syndrome berechnen
          LD     D,0        ; Syndrom zu
          LD     E,A        ; Relativadresse machen
          LD     HL,DMASK   ; Zeiger auf Decodier-Masken
          ADD    HL,DE      ; Zeiger auf richtige
                            ; Decodier-Maske
          LD     A,C        ; Datenbits holen
```

```
                AND     00001111B   ; Pruefbits wegmaskieren
                XOR     (HL)        ; Korrektur ausfuehren
                RET

; Masken zur Syndrom-Berechnung

SMASK:          DEFB    1011B       ; fuer Datenbit 0
                DEFB    0111B       ; fuer Datenbit 1
                DEFB    1101B       ; fuer Datenbit 2
                DEFB    1110B       ; fuer Datenbit 3
                DEFB    0001B       ; fuer Datenbit 4
                DEFB    0010B       ; fuer Datenbit 5
                DEFB    0100B       ; fuer Datenbit 6
                DEFB    1000B       ; fuer Datenbit 7

; Masken zur Decodierung

DMASK:          DEFB    00000000B   ; fuer Syndrom 0
                DEFB    00000000B   ; fuer Syndrom 1
                DEFB    00000000B   ; fuer Syndrom 2
                DEFB    10000000B   ; fuer Syndrom 3
                DEFB    00000000B   ; fuer Syndrom 4
                DEFB    10000000B   ; fuer Syndrom 5
                DEFB    10000000B   ; fuer Syndrom 6
                DEFB    00000010B   ; fuer Syndrom 7
                DEFB    00000000B   ; fuer Syndrom 8
                DEFB    10000000B   ; fuer Syndrom 9
                DEFB    10000000B   ; fuer Syndrom 10
                DEFB    00000001B   ; fuer Syndrom 11
                DEFB    10000000B   ; fuer Syndrom 12
                DEFB    00000100B   ; fuer Syndrom 13
                DEFB    00001000B   ; fuer Syndrom 14
                DEFB    10000000B   ; fuer Syndrom 15
```

Das Verfahren läßt sich verallgemeinern auf 2^r-r-1 Informationsbits und $r+1$ Prüfbits beziehungsweise Syndrome, wobei r eine positive ganze Zahl größer als 3 ist.

29.4 Rastergraphik

Viele Computersysteme verfügen über Möglichkeiten zur Darstellung von Bildern, die aus einzelnen Bildpunkten aufgebaut sind; man spricht in diesem Zusammenhang von *Rastergraphik*, weil die Bilder gerastert sind.

Die Verarbeitung von Bildern kann bei speicherorientierter Ein-/Ausgabe direkt auf dem zugeordneten Speicherbereich erfolgen (siehe Kapitel »Ein-/Ausgabe-Techniken«); andernfalls legen wir uns ein entsprechendes Speicherabbild an, auf dem wir arbeiten können. Wenn wir mit monochromen Bildern hantieren, also Bildern ohne Farben, genügt für die Darstellung eines Bildpunkts (engl. pixel) ein Bit-Speicher.

Da der Speicher byte-weise organisiert ist, gibt es zwei wesentlich verschiedene Möglichkeiten, wie das Paar (X,Y) aus horizontaler und vertikaler Punktkoordinate auf das Paar (a,b) aus Byte-Adresse und Bit-Adresse abgebildet werden kann:

1. Zeilenorganisation: Die Bit-Adresse ist gleichläufig mit der X-Koordinate; eine Erhöhung von X um 1 führt zu einer Erhöhung der Bit-Adresse um 1, modulo 8 gerechnet.

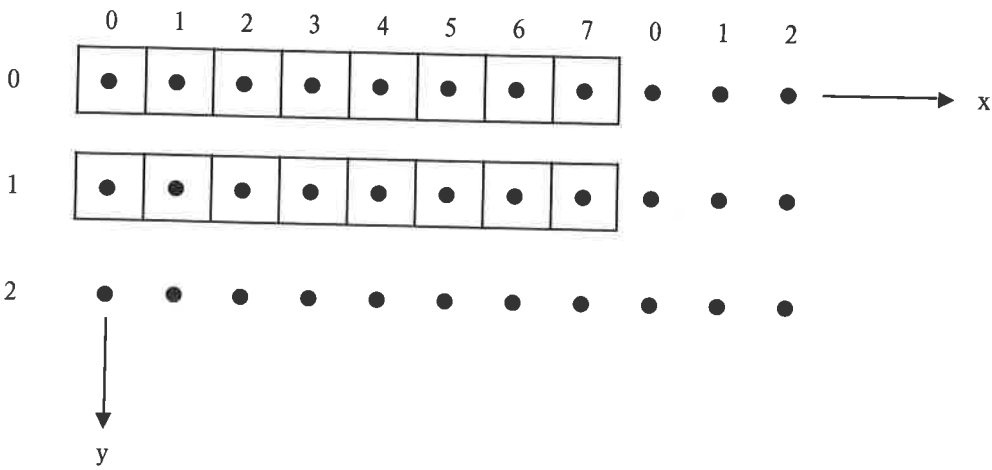


Bild 29.1. *Zeilenorganisation bei Bildern*

2. Spaltenorganisation: Die Bit-Adresse ist gleichläufig mit der Y-Koordinate; eine Erhöhung von Y um 1 führt zu einer Erhöhung der Bit-Adresse um 1, modulo 8 gerechnet (Bild 29.2).

Während die Zeilenorganisation bei Bildschirmen üblich ist, verwendet man Spaltenorganisation zum Beispiel bei Matrixdruckern.

Die X-Koordinate kann sich im Bereich 0 <= X < llx bewegen, die Y-Koordinate im Bereich 0 <= Y < lly. Der Einfachheit halber wollen wir annehmen, daß llx und lly durch 8 teilbar sind; wir führen deshalb noch die Größen lx = llx / 8 und ly = lly / 8 ein.

Typische Bildschirmabmessungen sind:

lx = 64, ly = 16
lx = 80, ly = 24
lx = 80, ly = 25

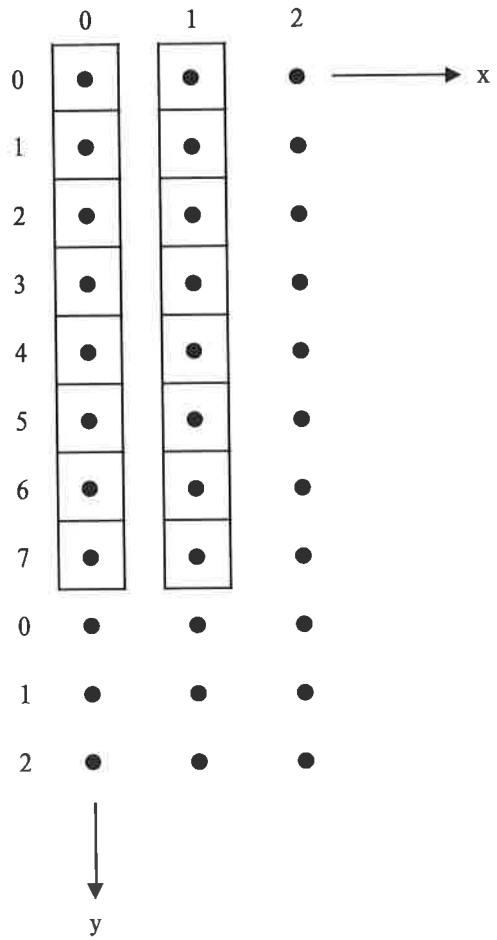


Bild 29.2. *Spaltenorganisation bei Bildern*

Für Matrixdrucker ist häufig anzutreffen:

lx = 80
lx = 132

Die Speicherabbildungsfunktionen (X,Y) –> (a,b) ergeben sich wie folgt:

1. Bei Zeilenorganisation:
   a = Y * lx + [X/8]
   b = X mod 8

2. Bei Spaltenorganisation:
   $a = \lfloor Y/8 \rfloor * llx + X$
   $b = Y \bmod 8$

Das Klammerpaar $\lfloor\ \rfloor$ (Gaußsche Klammern) steht für die Rundung zur nächstkleineren ganzen Zahl, also zum Beispiel $\lfloor 3.8 \rfloor = 3$, $\lfloor 2.0 \rfloor = 2$.

Wir schreiben uns als erstes Unterprogramme ZEIPIX und SPAPIX, welche uns zu vorgegebenen Koordinaten X und Y (im DE- und BC-Register) und fester Basis-Adresse des Pixel-Felds (im HL-Register) die Byte-Adresse a (im HL-Register) und die Bit-Adresse b (im A-Register) berechnen. Die Größen llx, lx, lly, ly sollen Konstanten sein.

Wir erinnern uns daran, daß ganzzahlige Division durch eine Zweierpotenz 2^r und Restbildung simultan durch die entsprechende Anzahl r von Rechtsverschiebungen realisiert werden können.

Die Multiplikationen führen wir mit den Multiplikationsroutinen ZEIMUL und SPAMUL durch. Da jeweils ein Operand fest und bekannt ist, wählen wir gestreckte Multiplikation, um die Geschwindigkeit zu erhöhen. Den variablen Operanden übergeben wir stets im BC-Register, das Ergebnis fällt im HL-Register an.

Wir sehen uns als Beispiel die Multiplikationsprogramme für lx = 80 (llx = 640) an; diese Daten treffen auf viele Bildschirme und Drucker zu.

```
ZEIMUL: LD      HL,0        ; 0
        ADD     HL,BC       ; 1
        ADD     HL,HL       ; 2
        ADD     HL,HL       ; 4
        ADD     HL,BC       ; 5
        ADD     HL,HL       ; 10
        ADD     HL,HL       ; 20
        ADD     HL,HL       ; 40
        ADD     HL,HL       ; 80
        RET

SPAMUL: CALL    ZEIMUL      ; 80
        ADD     HL,HL       ; 160
        ADD     HL,HL       ; 320
        ADD     HL,HL       ; 640
        RET
```

ZEIPIX und SPAPIX unterscheiden sich nur durch das Registerpaar, das verschoben wird, und das aufgerufene Multiplikationsprogramm:

```
ZEIPIX: XOR     A           ; zur Aufnahme der Bit-Adresse
                            ; vorbereiten
        SRL     D           ; Superregister
```

```
        RR      E       ; D&E&A
        RRA             ; rechtsverschieben
        SRL     D       ; Superregister
        RR      E       ; D&E&A
        RRA             ; rechtsverschieben
        SRL     D       ; Superregister
        RR      E       ; D&E&A
        RRA             ; rechtsverschieben
        RRCA            ; Bit-Adresse
        RRCA            ; durch
        RRCA            ; 5 Rechtsverschiebungen
        RRCA            ; rechtsbuendig
        RRCA            ; machen
        ADD     HL,DE   ; Relativadresse zu
                        ; Basis-Adresse addieren
        EX      DE,HL   ; und Ergebnis sichern
        CALL    ZEIMUL  ; Multiplikation durchfuehren
        ADD     HL,DE   ; Byte-Adresse berechnen
        RET

SPAPIX: XOR     A       ; zur Aufnahme der Bit-Adresse
                        ; vorbereiten
        SRL     B       ; Superregister
        RR      C       ; B&C&A
        RRA             ; rechtsverschieben
        SRL     B       ; Superregister
        RR      C       ; B&C&A
        RRA             ; rechtsverschieben
        SRL     B       ; Superregister
        RR      C       ; B&C&A
        RRA             ; rechtsverschieben
        RRCA            ; Bit-Adresse
        RRCA            ; durch
        RRCA            ; 5 Rechtsverschiebungen
        RRCA            ; rechtsbuendig
        RRCA            ; machen
        ADD     HL,DE   ; Relativadresse zu
                        ; Basis-Adresse addieren
        EX      DE,HL   ; und Ergebnis sichern
        CALL    SPAMUL  ; Multiplikation durchfuehren
        ADD     HL,DE   ; Byte-Adresse berechnen
        RET
```

Als nächstes schreiben wir ein Unterprogramm, das aus der Bit-Adresse eine Maske generiert, welche genau an der Stelle des adressierten Bits eine Eins stehen hat. Das HL-Register soll bei dieser Operation intakt bleiben, das A-Register trägt zunächst die Bit-Adresse, später die Maske.

```
MASKE:  EX    DE,HL       ; HL-Register sichern
        LD    HL,MASKEN   ; Basis-Adresse des Masken-Felds
        LD    B,0         ; Bit-Adresse zu
        LD    C,A         ; Wort machen
        ADD   HL,BC       ; Adresse der Maske berechnen
        LD    A,(HL)      ; Maske holen
        EX    DE,HL       ; HL-Register restaurieren
        RET

MASKEN: DEFB  00000001B   ; Maske fuer Bit 0
        DEFB  00000010B   ; Maske fuer Bit 1
        DEFB  00000100B   ; Maske fuer Bit 2
        DEFB  00001000B   ; Maske fuer Bit 3
        DEFB  00010000B   ; Maske fuer Bit 4
        DEFB  00100000B   ; Maske fuer Bit 5
        DEFB  01000000B   ; Maske fuer Bit 6
        DEFB  10000000B   ; Maske fuer Bit 7
```

Nun können wir zwei Unterprogramme ZEIMAS und SPAMAS schreiben, die uns aus den Koordinaten (im DE- und BC-Register) direkt die Byte-Adresse (im HL-Register) und die Bit-Maske (im A-Register) berechnen:

```
ZEIMAS: CALL  ZEIPIX      ; Byte-Adresse und
                          ; Bit-Adresse berechnen
        CALL  MASKE       ; Maske aus Bit-Adresse
                          ; generieren
        RET

SPAMAS: CALL  SPAPIX      ; Byte-Adresse und
                          ; Bit-Adresse berechnen
        CALL  MASKE       ; Maske aus Bit-Adresse
                          ; generieren
        RET
```

Nun folgen je vier Unterprogramme für das Setzen, Löschen, Invertieren und Testen eines Pixels (Koordinaten in DE und BC):

```
ZEISET:  CALL    ZEIMAS   ; Byte-Adresse und
                          ; Bit-Maske berechnen
         OR      (HL)     ; Bit setzen
         LD      (HL),A   ; Byte zurueckschreiben
         RET

ZEIRES:  CALL    ZEIMAS   ; Byte-Adresse und
                          ; Bit-Maske berechnen
         CPL               ; Maske invertieren
         AND     (HL)     ; Bit loeschen
         LD      (HL),A   ; Byte zurueckschreiben
         RET

ZEIINV:  CALL    ZEIMAS   ; Byte-Adresse und
                          ; Bit-Maske berechnen
         XOR     (HL)     ; Bit invertieren
         LD      (HL),A   ; Byte zurueckschreiben
         RET

ZEITST:  CALL    ZEIMAS   ; Byte-Adresse und
                          ; Bit-Maske berechnen
         AND     (HL)     ; Bit testen
         RET

SPASET:  CALL    SPAMAS   ; Byte-Adresse und
                          ; Bit-Maske berechnen
         OR      (HL)     ; Bit setzen
         LD      (HL),A   ; Byte zurueckschreiben
         RET

SPARES:  CALL    SPAMAS   ; Byte-Adresse und
                          ; Bit-Maske berechnen
         CPL               ; Maske invertieren
         AND     (HL)     ; Bit loeschen
         LD      (HL),A   ; Byte zurueckschreiben
         RET

SPAINV:  CALL    SPAMAS   ; Byte-Adresse und
                          ; Bit-Maske berechnen
         XOR     (HL)     ; Bit invertieren
         LD      (HL),A   ; Byte zurueckschreiben
         RET
```

```
SPATST:   CALL    SPAMAS    ; Byte-Adresse und
                            ; Bit-Maske berechnen
          AND     (HL)      ; Bit testen
          RET
```

Um unser Paket von Routinen noch schneller zu machen, führen wir zwei Optimierungen durch:

Um die Bit-Adresse rechtsbündig zu machen, mußten wir das A-Register fünfmal rechtsverschieben. Die Bit-Adresse wird aber nur gebraucht, um daraus die Bit-Maske zu generieren. Wir schieben die Bits der Bit-Adresse deshalb von rechts ins A-Register, wodurch sich eine zur eigentlichen Bit-Adresse bezüglich Bit 1 spiegelbildliche Pseudo-Bit-Adresse ergibt. Wir brauchen nun nur noch die Masken entsprechend umzuordnen, um die Pseudo-Bit-Adresse genauso wie die Bit-Adresse benutzen zu können.

Das Aufrufen von Unterprogrammen kostet relativ viel Zeit. Wir kopieren deshalb den Code möglichst vieler Unterprogramme an der Aufrufstelle ein. Es ergibt sich zum Beispiel folgendes Programmpaket:

```
MASKEN:   DEFB    00000001B   ; Maske fuer Bit 0
          DEFB    00010000B   ; Maske fuer Bit 4
          DEFB    00000100B   ; Maske fuer Bit 2
          DEFB    01000000B   ; Maske fuer Bit 6
          DEFB    00000010B   ; Maske fuer Bit 1
          DEFB    00100000B   ; Maske fuer Bit 5
          DEFB    00001000B   ; Maske fuer Bit 3
          DEFB    10000000B   ; Maske fuer Bit 7

ZEIMAS:
ZEIPIX:   XOR     A           ; zur Aufnahme der
                              ; Pseudo-Bit-Adresse
                              ; vorbereiten
          SRL     D           ; DE-Register
          RR      E           ; rechtsverschieben
          RLA                 ; Bit in Pseudo-Bit-Adresse
                              ; schieben
          SRL     D           ; DE-Register
          RR      E           ; rechtsverschieben
          RLA                 ; Bit in Pseudo-Bit-Adresse
                              ; schieben
          SRL     D           ; DE-Register
          RR      E           ; rechtsverschieben
          RLA                 ; Bit in Pseudo-Bit-Adresse
                              ; schieben
          ADD     HL,DE       ; Relativadresse zu
```

```
                                        ; Basis-Adresse addieren
        EX      DE,HL                   ; und Ergebnis sichern
ZEIMUL: LD      HL,0                    ; 0
        ADD     HL,BC                   ; 1
        ADD     HL,HL                   ; 2
        ADD     HL,HL                   ; 4
        ADD     HL,BC                   ; 5
        ADD     HL,HL                   ; 10
        ADD     HL,HL                   ; 20
        ADD     HL,HL                   ; 40
        ADD     HL,HL                   ; 80
        ADD     HL,DE                   ; Byte-Adresse berechnen
ZMASKE: EX      DE,HL                   ; HL-Register sichern
        LD      HL,MASKEN               ; Basis-Adresse des Masken-Felds
        LD      B,0                     ; Bit-Adresse zu
        LD      C,A                     ; Wort machen
        ADD     HL,BC                   ; Adresse der Maske berechnen
        LD      A,(HL)                  ; Maske holen
        EX      DE,HL                   ; HL-Register restaurieren
        RET

SPAMAS:
SPAPIX: XOR     A                       ; zur Aufnahme der
                                        ; Pseudo-Bit-Adresse
                                        ; vorbereiten
        SRL     B                       ; BC-Register
        RR      C                       ; rechtsverschieben
        RLA                             ; Bit in Pseudo-Bit-Adresse
                                        ; schieben
        SRL     B                       ; BC-Register
        RR      C                       ; rechtsverschieben
        RLA                             ; Bit in Pseudo-Bit-Adresse
                                        ; schieben
        SRL     B                       ; BC-Register
        RR      C                       ; rechtsverschieben
        RLA                             ; Bit in Pseudo-Bit-Adresse
                                        ; schieben
        ADD     HL,DE                   ; Relativadresse zu
                                        ; Basis-Adresse addieren
        EX      DE,HL                   ; und Ergebnis sichern
SPAMUL: LD      HL,0                    ; 0
        ADD     HL,BC                   ; 1
        ADD     HL,HL                   ; 2
```

```
        ADD     HL,HL       ; 4
        ADD     HL,BC       ; 5
        ADD     HL,HL       ; 10
        ADD     HL,HL       ; 20
        ADD     HL,HL       ; 40
        ADD     HL,HL       ; 80
        ADD     HL,HL       ; 160
        ADD     HL,HL       ; 320
        ADD     HL,HL       ; 640
        ADD     HL,DE       ; Byte-Adresse berechnen
SMASKE: EX      DE,HL       ; HL-Register sichern
        LD      HL,MASKEN   ; Basis-Adresse des Masken-Felds
        LD      B,0         ; Bit-Adresse zu
        LD      C,A         ; Wort machen
        ADD     HL,BC       ; Adresse der Maske berechnen
        LD      A,(HL)      ; Maske holen
        EX      DE,HL       ; HL-Register restaurieren
        RET

ZEISET: CALL    ZEIMAS      ; Byte-Adresse und
                            ; Bit-Maske berechnen
        OR      (HL)        ; Bit setzen
        LD      (HL),A      ; Byte zurueckschreiben
        RET

ZEIRES: CALL    ZEIMAS      ; Byte-Adresse und
                            ; Bit-Maske berechnen
        CPL                 ; Maske invertieren
        AND     (HL)        ; Bit loeschen
        LD      (HL),A      ; Byte zurueckschreiben
        RET

ZEIINV: CALL    ZEIMAS      ; Byte-Adresse und
                            ; Bit-Maske berechnen
        XOR     (HL)        ; Bit invertieren
        LD      (HL),A      ; Byte zurueckschreiben
        RET

ZEITST: CALL    ZEIMAS      ; Byte-Adresse und
                            ; Bit-Maske berechnen
        AND     (HL)        ; Bit testen
        RET
```

```
SPASET:  CALL   SPAMAS   ; Byte-Adresse und
                         ; Bit-Maske berechnen
         OR     (HL)     ; Bit setzen
         LD     (HL),A   ; Byte zurueckschreiben
         RET

SPARES:  CALL   SPAMAS   ; Byte-Adresse und
                         ; Bit-Maske berechnen
         CPL             ; Maske invertieren
         AND    (HL)     ; Bit loeschen
         LD     (HL),A   ; Byte zurueckschreiben
         RET

SPAINV:  CALL   SPAMAS   ; Byte-Adresse und
                         ; Bit-Maske berechnen
         XOR    (HL)     ; Bit invertieren
         LD     (HL),A   ; Byte zurueckschreiben
         RET

SPATST:  CALL   SPAMAS   ; Byte-Adresse und
                         ; Bit-Maske berechnen
         AND    (HL)     ; Bit testen
         RET
```

Beim Zeichnen von Kurven entsteht oft das Problem, daß die Rasterung zu grob für eine exakte Darstellung der Kurve ist. Wir *approximieren* dann die Punkte der Kurve durch möglichst nahe gelegene darstellbare Punkte. Die Algorithmen zur Berechnung dieser Punkte sind zum Teil sehr trickreich. Wir sehen uns einen Algorithmus zum Zeichnen der Strecke von (X_1,Y_1) bis (X_2,Y_2) an:

```
dx <- X2 - X1
dy <- Y2 - Y1
dt <- max (|dx|, |dy|)
X <- X1
Y <- Y1
p <- 0
q <- 0
Zeichne (X,Y)
wiederhole
          <- p + |dx|
          wenn      2 * p > dt
          dann      p <- p - dt
                    X <- X + sign (dx)
          q <- q + |dy|
```

```
wenn        2 * q > dt
dann        q <– q – dt
            Y <– Y + sign (dy)
Zeichne (X,Y)
```

für t von 1 **bis** dt **in Schritten von** 1

Betrachte dazu folgendes Beispiel: Zu zeichnen ist die Strecke von (0,0) bis (2,5). Es gilt:

dx = 2
dy = 5
dt = 5

t	X	Y	p	q
0	0	0	0	0
1	0	1	2	0
2	1	2	–1	0
3	1	3	1	0
4	2	4	–2	0
5	2	5	0	0

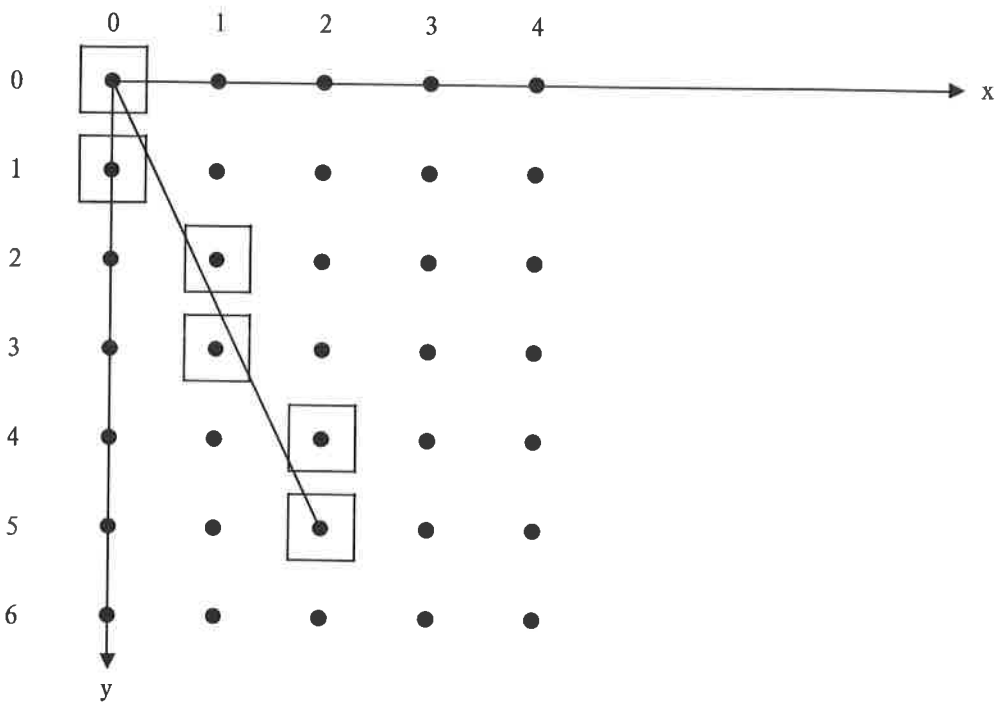


Bild 29.3. *Approximieren einer Strecke*

Bei der Durchführung des Verfahrens müssen wir beachten, daß die Größen dx und dy negativ sein können.

Wir wählen folgende Registerbelegung:

X	DE
Y	BC
|dx|	DE'
|dy|	BC'
p	IX
q	IY
−dt	HL'
t	HL
sign (dx)	A'_0
sign (dy)	A'_1

Statt des Signums von dx und dy speichern wir nur das Vorzeichen; es steht also 0 für Null und positives dx beziehungsweise dy, 1 für negatives dx beziehungsweise dy.

Zu Beginn des Verfahrens müssen X_1 und Y_1 im DE- und BC-Register, X_2 und Y_2 im DE'- und BC'-Register stehen. Bei der konkreten Realisierung nehmen wir an, daß die zu zeichnenden Punkte gesetzt werden sollen und daß Zeilenorganisation vorliegt. Das Programm lautet damit:

```
STRECK:  EXX                ; X2 in
         PUSH    DE         ; primaeres
         EXX                ; Register
         POP     HL         ; uebertragen
         XOR     A          ; Vorzeichen positiv vorbesetzen,
                            ; Uebertrag-Flag loeschen
         SBC     HL,DE      ; dx berechnen
         JP      NC,XPOSIT  ; dx >= 0
         INC     A          ; negatives Vorzeichen eintragen
         EX      AF,AF'     ; Vorzeichen sichern
         LD      A,H        ; Betrag
         CPL                ; von
         LD      H,A        ; dx
         LD      A,L        ; durch
         CPL                ; Invertieren
         LD      L,A        ; von HL
         INC     HL         ; berechnen
         EX      AF,AF'     ; Vorzeichen holen
XPOSIT:  PUSH    HL         ; |dx| in
         EXX                ; sekundaeres Register
         POP     DE         ; uebertragen
```

```
        PUSH    BC          ; Y2 in
        EXX                 ; primaeres Register
        POP     HL          ; uebertragen
        OR      A           ; Uebertrag-Flag loeschen
        SBC     HL,BC       ; dy berechnen
        JP      NC,YPOSIT   ; dy >= 0
        SET     1,A         ; negatives Vorzeichen eintragen
        EX      AF,AF'      ; Vorzeichen sichern
        LD      A,H         ; Betrag
        CPL                 ; von
        LD      H,A         ; dy
        LD      A,L         ; durch
        CPL                 ; Invertieren
        LD      L,A         ; von HL
        INC     HL          ; berechnen
        EX      AF,AF'      ; Vorzeichen holen
YPOSIT: PUSH    HL          ; |dy|in
        EXX                 ; sekundaeres Register
        POP     BC          ; uebertragen
        LD      H,D         ; |dx|
        LD      L,E         ; kopieren
        OR      A           ; Uebertrag-Flag loeschen
        SBC     HL,BC       ; |dx|+ |dy| berechnen
        LD      H,B         ; |dy|
        LD      L,C         ; kopieren
        P       M,NEGAT     ; |dx|< |dy|, dt = |dy|
        LD      H,D         ; |dx|
        LD      L,E         ; kopieren
NEGAT:  PUSH    HL          ; t sichern
        EX      AF,AF'      ; Vorzeichen sichern
        LD      A,H         ; Negation
        CPL                 ; von
        LD      H,A         ; dt
        LD      A,L         ; durch
        CPL                 ; Invertieren
        LD      L,A         ; von HL
        INC     HL          ; berechnen
        EXX                 ; t in primaeres
        POP     HL          ; Register uebertragen
        LD      IX,0        ; p initialisieren
        LD      IY,0        ; q initialisieren
        PUSH    HL          ; Zaehler sichern
        PUSH    BC          ; X und Y
```

```
        PUSH    DE          ; retten
        CALL    ZEISET      ; (X,Y) zeichnen
        POP     DE          ; X und Y
        POP     BC          ; restaurieren
        POP     HL          ; Zaehler restaurieren
TEST:   LD      A,H         ; Zaehler auf
        OR      L           ; Null testen
        RET     Z           ; Strecke gezeichnet
        DEC     HL          ; Zaehler um 1 vermindern
        EXX                 ; sekundaeren Registersatz holen
        ADD     IX,DE       ; p <- p + |dx|
        PUSH    IX          ; p sichern
        ADD     IX,IX       ; 2 * p berechnen
        EX      DE,HL       ; Berechnung
        ADD     IX,DE       ; von
        EX      DE,HL       ; 2 * p - dt
        PUSH    IX          ; 2 * p - dt sichern
        EX      (SP),HL     ; - dt sichern,
                            ; 2 * p - dt holen
        DEC     HL          ; 2 * p - dt - 1 berechnen
        BIT     7,H         ; 2 * p > dt testen
        POP     HL          ; - dt restaurieren
        POP     IX          ; p restaurieren
        JP      NZ,XNICHT   ; nichts zu tun
        EX      DE,HL       ; Berechnung
        ADD     IX,DE       ; von
        EX      DE,HL       ; p <- p - dt
        EXX                 ; primaeren Registersatz holen
        INC     DE          ; positives Vorzeichen
                            ; von dx annehmen
        EX      AF,AF'      ; Vorzeichen holen
        BIT     0,A         ; Vorzeichen testen
        JP      Z,XPOS      ; dx ist positiv
        DEC     DE          ; Korrektur
        DEC     DE          ; X bewegen
XPOS:   EXX                 ; sekundaeren Registersatz holen
        EX      AF,AF'      ; Vorzeichen sichern
XNICHT: ADD     IY,BC       ; q <- q + |dy|
        PUSH    IY          ; q sichern
        ADD     IY,IY       ; 2 * q berechnen
        X       DE,HL       ; Berechnung
        ADD     IY,DE       ; von
        EX      DE,HL       ; 2 * q - dt
```

```
        PUSH    IY          ; 2 * q - dt sichern
        EX      (SP),HL     ; - dt sichern,
                            ; 2 * p - dt holen
        DEC     HL          ; 2 * q - dt - 1 berechnen
        BIT     7,H         ; 2 * q > dt testen
        POP     HL          ; - dt restaurieren
        POP     IY          ; q restaurieren
        P       NZ,YNICHT   ; nichts zu tun
        EX      DE,HL       ; Berechnung
        ADD     IY,DE       ; von
        EX      DE,HL       ; q <- q - dt
        EXX                 ; primaeren Registersatz holen
        INC     BC          ; positives Vorzeichen
                            ; von dy annehmen
        EX      AF,AF'      ; Vorzeichen holen
        BIT     1,A         ; Vorzeichen testen
        JP      Z,YPOS      ; dy ist positiv
        DEC     BC          ; Korrektur
        DEC     BC          ; Y bewegen
YPOS:   EXX                 ; sekundaeren Registersatz holen
        EX      AF,AF'      ; Vorzeichen sichern
YNICHT: EXX                 ; primaeren Registersatz holen
        PUSH    HL          ; Zaehler sichern
        PUSH    BC          ; X und Y
        PUSH    DE          ; retten
        CALL    ZEISET      ; (X,Y) zeichnen
        POP     DE          ; X und Y
        POP     BC          ; restaurieren
        POP     HL          ; Zaehler restaurieren
        JP      TEST        ; gesamte Strecke zeichnen
```

29.5 Backtracking

Backtracking ist eine Standardtechnik aus dem Bereich der sogenannten *Künstlichen Intelligenz*, KI (engl. artificial intelligence, AI). Unter Künstlicher Intelligenz stellte man sich in den sechziger Jahren Methoden vor, welche zur Programmierung lernender Computer tauglich sein sollten. Mittlerweile ist man etwas bescheidener geworden und versteht unter dem Gebiet der Künstlichen Intelligenz bestimmte Formen des computerisierten Problemlösens, die sich auf exakte Methoden und auf Faustregeln (Heuristiken) abstützen; ein Beispiel für derartige Computersysteme sind sog. *Expertensysteme.*

Backtracking dient der systematischen Auffindung einer, mehrerer oder aller Lösungen bestimmter kombinatorischer Probleme. Es gibt eine unübersehbare Zahl solcher Probleme,

die mittels Backtracking gelöst werden können. Einige Beispiele: Auf wie viele verschiedene Weisen kann ein vorgelegtes Wort aus einer vorgegebenen Menge nicht notwendig verschiedener Buchstaben gebildet werden? Können 8 Damen auf einem Schachbrett so aufgestellt werden, daß sie sich gegenseitig nicht bedrohen? Wie viele verschiedene Resultate kann man aus einer Menge von Zahlen durch Anwendung von Addition und Subtraktion erzeugen? Wie oft läßt sich eine bestimmte positive ganze Zahl als Summe mehrerer verschiedener positiver ganzer Zahlen darstellen? Kann man aus einer Menge vorgelegter Bausteine eine bestimmte Figur aufbauen? ...

Die allgemeine Vorgehensweise beim Backtracking ist folgende: Man gibt sich eine endliche Folge von Variablen vor. Jede dieser Variablen kann mit einer endlichen Menge von numerischen Werten belegt werden. Jede Belegung der Variablen mit Werten korrespondiert zu einer konkreten Anordnung des Problems; diese Anordnung kann die Problemvorgaben erfüllen (zulässige Lösung) oder verletzen (unzulässige Lösung). Für jede der Variablen existiert eine Regel, die besagt, ob eine Erhöhung des Werts dieser Variablen unter gleichzeitigem Festhalten der Werte aller vorhergehender Variablen prinzipiell wieder zu zulässigen Lösungen führen kann oder nicht.

Da man den Wertebereich einer Variablen linear ordnen kann, ist auf der Menge der Anordnungen eine lexikalische Ordnung definiert: Eine Anordnung steht lexikalisch vor einer anderen Anordnung, wenn dies für die zugehörigen Folgen von Variablenwerten gilt (bezüglich lexikalischer Ordnung vergleiche das Kapitel »Zeichenketten«).

Wir durchlaufen nun die Menge der zulässigen Lösungen (und eine Teilmenge der unzulässigen Lösungen) mittels des folgenden Algorithmus:

1. Zunächst erhalten alle Variablen ihren jeweils kleinsten Wert (Startanordnung).
2. Ausgehend von einer bestimmten Variablenbelegung wird eine lexikalisch folgende Variablenbelegung bestimmt, falls eine solche existiert. Dazu stellt man fest, ob die Regel für die letzte Variable besagt, daß eine Erhöhung des Werts zu keiner zulässigen Lösung mehr führen kann. Ist dies nicht der Fall, so wird der Wert der Variablen auf den nächstgrößeren Wert erhöht. Andernfalls wird dieselbe Untersuchung für die vorletzte Variable durchgeführt und so weiter. Führt auch eine Erhöhung der ersten Variablen nicht mehr auf zulässige Lösungen, so endet das Verfahren. Nach Erhöhung des Werts einer Variablen wird den darauf folgenden Variablen der jeweils kleinste Wert ihres Wertebereichs zugewiesen.

Die Regeln garantieren, daß nur unzulässige Lösungen ausgelassen werden. Das Verfahren findet also alle zulässigen Lösungen. Eine triviale Regel für eine beliebige Variable wäre: Eine Erhöhung des Werts über den Wertebereich hinaus kann keine zulässige Lösung mehr liefern.

Wir veranschaulichen uns diese Technik an einem konkreten Beispiel:

Gesucht wird die Anzahl t der Möglichkeiten, eine positive ganze Zahl m als Summe einer festen Anzahl n von positiven ganzen Zahlen darzustellen (Partitionierungsproblem). Die Reihenfolge der Summanden soll dabei eine Rolle spielen; die Zerlegungen $4 = 1 + 3$ und $4 = 3 + 1$ sind in diesem Sinne verschieden.

Zunächst erledigen wir die Trivialfälle: Gilt $m < n$ oder $n = 0$, so ist $t = 0$; gilt $n = 1$, so ist $t = 1$. Wir nehmen im folgenden stets $2 <= n <= m$ an.

Wir geben uns n Variablen $v_1, ..., v_n$ vor, deren Wertebereich jeweils aus den ganzen Zahlen von 1 bis m besteht. Für die Variable v_i besteht die Regel

R_i: Eine Erhöhung von v_i liefert keine zulässige Lösung mehr, wenn die Summe der Variablen v_1 bis v_i den Wert m−n+i−1 übersteigt.

Dies erklärt sich aus der Tatsache, daß die Variablen v_{i+1} bis v_n jeweils mindestens den Wert 1 besitzen, die Summe aller Variablen aber nicht größer als m sein darf.

Der unserem Problem angepaßte Backtracking-Algorithmus lautet also:

```
Setze die Anzahl t der gefundenen Belegungen auf 0
Initialisiere alle Variablen v1 bis vn mit 1
wiederhole
        wenn    Belegung zulässig
        dann    Erhöhe die Anzahl t der gefundenen Belegungen um 1
        Initialisiere den Variablenindex i mit n
        wiederhole
                wenn    Summe der Variablen v1 bis vi > m−n+i−1
                dann    Vermindere Variablenindex i um 1
                sonst   Erhöhe die Variable vi um 1
                        Nachfolgerbelegung gefunden
        bis     Variablenindex i = 0
        oder    Nachfolgerbelegung gefunden
        wenn    Variablenindex i = 0
        dann    existiert keine Nachfolgerbelegung
        sonst   Initialisiere alle Variablen vi+1 bis vn mit 1
bis     keine Nachfolgerbelegung mehr existiert
```

Dieser Algorithmus läßt sich optimieren. Aus den Rahmenbedingungen des Verfahrens ist ersichtlich, daß zu vorgegebenen Werten der Variablen v_1 bis v_{n-1} höchstens ein Wert für v_n existiert, der in einer zulässigen Belegung resultiert. Wir durchlaufen deshalb nicht jedesmal den Wertebereich von v_n, sondern setzen v_n gezielt auf diesen eindeutig bestimmten Wert, falls die Belegung von v_1 bis v_{n-1} überhaupt noch eine zulässige Belegung erlaubt. Als Folge davon liefert das Verfahren nur noch zulässige Belegungen. Die Variable v_n können wir jetzt unterdrücken, da sie sich eindeutig aus den übrigen Variablen ergibt, wenn die Belegung zulässig sein soll. Der optimierte Algorithmus lautet damit:

```
Setze t auf 0
Initialisiere v1 bis vn−1 mit 1
wiederhole
        Erhöhe t um 1
        Initialisiere i mit n−1
        wiederhole
                wenn    Summe von v1 bis vi > m − n + i − 1
                dann    Vermindere i um 1
```

```
                sonst       Erhöhe v_i um 1
                            Nachfolgerbelegung gefunden
        bis     i = 0
        oder    Nachfolgerbelegung gefunden
        wenn    i = 0
        dann    existiert keine Nachfolgerbelegung
        sonst   Initialisiere v_{i+1} bis v_{n-1} mt 1
bis     keine Nachfolgerbelegung mehr existiert
```

Die Initialisierung der Variablen zu Beginn des Verfahrens läßt sich mit der Re-Initialisierung der Variablen v_i bis v_{n-1} zusammenfassen, wenn wir zu Beginn den Variablenindex i mit dem Wert 0 initialisieren. Fällt während der Suche nach einer Nachfolgerbelegung der Variablenindex i nochmals auf den Wert 0, so existiert keine Nachfolgerbelegung.

```
t <- 0
i <- 0
wiederhole
        t <- <t> + 1
        wiederhole
                i <- <i> + 1
                v_i <- 1
        solange <i> <= n - 2
        wiederhole
                wenn    Summe von <v_1> bis <v_i> > m - n + <i> - 1
                dann    i <- <i> - 1
                sonst   v_i <- <v_i> + 1
                        Nachfolgerbelegung gefunden
        bis     <i> = 0
        oder    Nachfolgerbelegung gefunden
bis     <i> = 0
```

Zur Vereinfachung der Durchführung des Algorithmus führen wir die Summe der Variablen v_1 bis v_i stets in einer Summenvariablen s mit. Nur wenn sich der Variablenindex i ändert, muß s neu berechnet werden.

```
t <- 0
i <- 0
s <- 0
wiederhole
        t <- <t> + 1
        wiederhole
                i <- <i> + 1
                v_i <- 1
                s <- <s> + 1
```

```
        solange <i><= n – 2
        wiederhole
                wenn    <s> > m – n + <i> – 1
                dann    s <– <s> – <v_i>
                        i <– <i> – 1
                sonst   v_i <– <v_i> + 1
                        s <– <s> + 1
                        Nachfolgerbelegung gefunden
        bis     <i> = 0
        oder    Nachfolgerbelegung gefunden
bis     <i> = 0
```

Auch die Neuberechnung der Größe m–n+i–1 kann vereinfacht werden; wir führen diese Größe in einer Variablen r mit und gleichen sie den Änderungen des Variablenindex i an:

```
t <– 0
i <– 0
s <– 0
r <– m – n – 1
wiederhole
        t <– <t> + 1
        wiederhole
                i <– <i> + 1
                v_i <– 1
                s <– <s> + 1
                r <– <r> + 1
        solange <i><= n – 2
        wiederhole
                wenn    <s> > <r>
                dann    s <– <s> – <v_i>
                        i <– <i> – 1
                        r <– <r> – 1
                sonst   v_i <– <v_i> + 1
                        s <– <s> + 1
                        Nachfolgerbelegung gefunden
        bis     <i> = 0
        oder    Nachfolgerbelegung gefunden
bis     <i> = 0
```

Nun kommen wir allmählich zu den Implementierungs-Entscheidungen; die Transformationen am Algorithmus werden dabei zunehmend maschinenbezogen.

Die Variablen v_1 bis v_{n-1} stellen wir am besten durch ein Feld von Variablen dar. Wenn wir dieses Feld so organisieren, wie wir es im Kapitel »Felder« getan haben, so kann jeder Bezug v_i

durch einen Zeiger z dargestellt werden. Bei Änderungen des Variablenindex i ändert sich auch der Zeiger z, der stets auf die Variable v_i gerichtet sein soll.

Wir konkretisieren jetzt unsere Anforderungen an die Konstanten m und n, und setzen stets die Beziehungen m,n < 256 voraus. Die ganzen Zahlen im Bereich 1 bis m lassen sich dann durch ein Byte darstellen. Wir teilen den Variablen v_1 bis v_{n-1} deshalb jeweils ein Byte Speicherplatz zu.

```
t <– 0
i <– 0
s <– 0
r <– m – n – 1
z <– Adresse der fiktiven Variablen v0
wiederhole
        t <– <t> + 1
        wiederhole
                i <– <i> + 1
                z <– <z> + 1
                (<z>) <– 1
                s <– <s> + 1
                r <– <r> + 1
        solange <i> <= n – 2
        wiederhole
                wenn    <s> > <r>
                dann    s <– <s> – <(<z>)>
                        i <– <i> – 1
                        z <– <z> – 1
                        r <– <r> – 1
                sonst   (<z>) <– <(<z>)> + 1
                        s <– <s> + 1
                        Nachfolgerbelegung gefunden
        bis     <i> = 0
        oder    Nachfolgerbelegung gefunden
bis     <i> = 0
```

Wir ordnen nun den einzelnen Konstanten und Variablen Register zu. Als Zeiger auf die Variablen v_1 bis v_{n-1} eignet sich am besten das HL-Register; wir nehmen an, daß dieses zu Beginn des Verfahrens auf die fiktive 0-te Komponente des Byte-Felds v_1 bis v_{n-1} zeigt. Die Anzahl der zulässigen Belegungen kann sehr groß werden. Wir wählen deshalb als Variable t die Registerkonkatenation HL'&DE'; falls auch dieses Superregister zu klein ist, erfolgt ein Abbruch des Verfahrens. Echte Arithmetik wird eigentlich nur auf der Variablen s getrieben; wir plazieren diese deshalb im A-Register. Um die beiden Größen s und r besser vergleichen zu können, empfiehlt es sich, statt r die Größe r'=r+1 mitzuführen; diese legen wir ins C-Register. Der Variablenindex i hat Zählcharakter; wir stellen deshalb dafür das B-Register bereit. Die Kon-

stanten m und n erwarten wir im D-Register beziehungsweise E-Register. Die Hilfsgröße n–2 halten wir im A'-Register. Zur Erhöhung des Superregisters HL'&DE' halten wir noch die Hilfsgröße 1 im BC'-Register.

s	A
i	B
r'	C
m	D
n	E
z	HL
n–2	A'
1	BC'
t	HL'&DE'

An einigen Stellen des Verfahrens führen wir noch lokale Optimierungen durch. Insbesondere beobachten wir, daß i genau dann den Wert 0 besitzt, wenn auch s den Wert 0 besitzt. Insgesamt erhalten wir folgenden Algorithmus mit konkreter Registerbelegung:

```
Unterprogramm    Backtr
        BC' <– 1
        DE' <– 0
        HL' <– 0
        wenn     <D><<E>
        dann     Verlasse Unterprogramm
        C <– <D> – <E>
        wenn     <E> = 1
        dann     DE' <– <DE'> + 1
                 Verlasse Unterprogramm
        wenn     <E> = 0
        dann     Verlasse Unterprogramm
        A' <– <E> – 2
        A <– 0
        B <– 0
        wiederhole
                DE' <– <DE'> + <BC'>
                wenn     Übertrag
                dann     HL' <– <HL'> + <BC'>
                         wenn     Übertrag
                         dann     Abbruch des Verfahrens
                wiederhole
                         B <– <B> + 1
                         HL <– <HL> + 1
                         (<HL>) <– 1
```

```
                    A <- <A> + 1
                    C <- <C> + 1
              solange <B> <= <A'>
              wiederhole
                    wenn    <A> >= <C>
                    dann    A <- <A> - <(<HL>)>
                            HL <- <HL> - 1
                            C <- <C> - 1
                            B <- <B> - 1
                    sonst   (<HL>) <- <(<HL>)> + 1
                            A <- <A> + 1
                            Nachfolgerbelegung gefunden
              bis   <B> = 0
              oder  Nachfolgerbelegung gefunden
        bis   <A> = 0
Ende Unterprogramm
```

Dieser Algorithmus läßt sich nun fast schematisch in ein Programm umsetzen:

```
BACKTR:                                 ; Unterprogramm  Backtr
        LD      BC,1                    ; BC' <- 1
        LD      D,B                     ; DE' <- 0
        LD      E,B
        LD      H,B                     ; HL' <- 0
        LD      L,B
        EXX
        LD      A,D                     ; wenn  <D> < <E>
        SUB     E
        RET     C                       ; dann Verlasse Unterprogramm
        LD      C,A                     ; C <- <D> - <E>
        LD      A,E                     ; wenn  <E> <> 1
        DEC     A
        JP      NZ,NICHT1               ; dann springe
        EXX                             ; DE' <- <DE'> + 1
        INC     DE
        RET                             ; Verlasse Unterprogramm
NICHT1:
        DEC     A                       ; wenn  <E> = 0
        RET     Z                       ; dann Verlasse Unterprogramm
        EX      AF,AF'                  ; A' <- <E> - 2
        XOR     A                       ; A <- 0
        LD      B,A                     ; B <- 0
ANORDN:                                 ; wiederhole
```

```
                EXX                             ; DE' <- <DE'> + <BC'>
                EX      DE,HL
                ADD     HL,BC
                EX      DE,HL
                JP      NC,KEINUE               ; wenn kein Uebertrag
                                                ; dann springe
                ADD     HL,BC                   ; HL' <- <HL'> + <BC'>
                JP      C,UEBERL                ; wenn Uebertrag
                                                ; dann Abbruch des Verfahrens
KEINUE:         EXX
FUELL:
                                                ; wiederhole
                EX      AF,AF'                  ; solange <B> <= <A'>
                CP      B
                JP      C,GEFUEL
                EX      AF,AF'
                INC     B                       ; B <- <B> + 1
                INC     HL                      ; HL <- <HL> + 1
                LD      (HL),1                  ; (<HL>) <- 1
                INC     A                       ; A <- <A> + 1
                INC     C                       ; C <- <C> + 1
                JP      FUELL
GEFUEL:         EX      AF,AF'
SUCHE:
                                                ; wiederhole
                CP      C                       ; wenn  <A> >= <C>
                JP      NC,SCHRIT               ; dann
                INC     (HL)                    ; sonst  (<HL>) <- <(<HL>)> + 1
                INC     A                       ; A <- <A> + 1
                JP      TEST                    ; Nachfolgerbelegung gefunden
SCHRIT:         SUB     (HL)                    ; dann  A <- <A> - <(<HL>)>
                DEC     HL                      ; HL <- <HL> - 1
                DEC     C                       ; C <- <C> - 1
                DJNZ    SUCHE                   ; B <- <B> - 1
                                                ; bis <B> = 0
                                                ; oder Nachfolgerbelegung gefunden
TEST:           OR      A                       ; bis <A> = 0
                JP      NZ,ANORDN
                RET                             ; Ende Unterprogramm
```

Wenn Sie das Beispiel richtig verstanden haben, wird Ihnen folgende Modifikation des Problems keine Schwierigkeiten bereiten:

Bestimmen Sie die Anzahl der Möglichkeiten, eine positive ganze Zahl m als Summe von n positiven ganzen Zahlen darzustellen, wobei kein Summand größer als l ist.

Lösungen zu den Übungen

Kapitel 2.1

1. Die Berechnungen lauten:

   1010B + 1010B = 10100B
   1110111B + 11011B = 10010010B
   101000B − 1010B = 11110B
   11010111B − 1101011B = 1101100B

2. Die Umwandlungen lauten:

   65 = 1000001B
   127 = 1111111B
   1194 = 10010101010B
   85 = 1010101B

3. Die dezimalen Werte zu Aufgabe 1 lauten:

   10 + 10 = 20
   119 + 27 = 146
   40 − 10 = 30
   215 − 107 = 108

Kapitel 2.2

1. Die Umwandlungen lauten:

   11111111B = FFH
   101100111B = 167H

1000001B = 41H
11011B = 1BH

2. Die Umwandlungen lauten:

80H = 10000000B
C4H = 11000100B
1234H = 1001000110100B
AAAAH = 1010101010101010B

3. Die Berechnungen lauten:

36H	FFH	ADACH	A000H
+ 14H	− C3H	+ E0FH	− 0E32H
4AH	3CH	BBBBH	91CEH

Kapitel 2.3

1. Die Umrechnungen lauten:

Dezimal	Hexadezimal	Oktal	Binär
36839	8FE7	107747	1000111111100111
42391	A597	122627	1010010110010111
6140	17FC	13774	0001011111111100
53670	D1A6	150646	1101000110100110

Kapitel 2.4

1. Die Darstellungen sind:

Dezimal	Binär
27233	0110101001100001
51896	1100101010111000
65983	nicht darstellbar
12356	0011000001000100

2. Die Darstellungen sind:

Dezimal	2-Komplement	1-Komplement	Vorzeichen-Betrag
92	01011100	01011100	01011100
-123	10000101	10000100	11111011
0	00000000	00000000	00000000
		11111111	10000000
-128	10000000	nicht darstellbar	nicht darstellbar
128	nicht darstellbar	nicht darstellbar	nicht darstellbar

Kapitel 6.1

1. Das Programm lautet:

```
LD      B,74H       ; B-Register mit 74H laden
```

2. Das Programm lautet:

```
LD      H,0F7H      ; H-Register mit F7H laden
```

   Haben Sie an die führende Null gedacht?

3. Der erste Operand eines LD-Befehls ist stets das Ziel der Transportoperation; er muß also eine Variable (Register oder Speicherzelle) sein.

4. Das Programm lautet:

```
LD      C,0D4H      ; C-Register mit D4H laden
```

5. Die drei Programme sind beispielsweise:

```
LD      E,96        ; E-Register mit 96 laden
LD      A,219Q      ; A-Register mit 219Q laden
LD      E,1101011B  ; E-Register mit 1101011B laden
```

Natürlich hätten wir die Zahlen auch ins Hexadezimalsystem umrechnen können; so geben sie aber die Vorgaben besser wieder.

Es ist eine gute Praxis, viele Kommentare zu schreiben, auch wenn das bei so kleinen Beispielen etwas übertrieben wirkt.

Kapitel 6.2

1. Die zugehörigen Programme lauten:

```
ADD     A,20H       ; Inhalt des A-Registers
                    ; um 20H erhoehen
ADD     A,48        ; Inhalt des A-Registers
                    ; um 48 erhoehen
ADD     A,221Q      ; Inhalt des A-Registers
                    ; um 221Q erhoehen
ADD     A,11010010B ; Inhalt des A-Registers
                    ; um 11010010B erhoehen
```

Den ersten Operanden der Addition müssen Sie zuvor ins A-Register bringen!

2. Die gesuchten Programme lauten:

```
SUB     20H         ; Inhalt des A-Registers
                    ; um 20H erniedrigen
SUB     48          ; Inhalt des A-Registers
                    ; um 48 erniedrigen
SUB     116Q        ; Inhalt des A-Registers
                    ; um 116Q erniedrigen
SUB     10100001B   ; Inhalt des A-Registers
                    ; um 10100001B erniedrigen
```

Den ersten Operanden der Subtraktion müssen Sie zuvor ins A-Register bringen!

3. Wir benötigen jeweils das gleiche Programm, nämlich

```
NEG                 ; Inhalt des A-Registers negieren
```

Die Wirkung sehen Sie in folgender Tabelle:

Wert	Negation	Z-Flag	S-Flag	C-Flag	P-Flag
80H	80H	0	1	1	1
0	0	1	0	0	0
164Q	214Q	0	1	1	0
10111001B	01000111B	0	0	1	0

Den Operanden der Negation müssen Sie zuvor ins A-Register bringen!

4. Das Subtraktions-Flag wird abhängig vom Befehl gesetzt oder rückgesetzt; es hängt niemals von den Operanden ab.

   Ist das Null-Flag gesetzt, so ist das Ergebnis der Operation Null, also nicht negativ; damit ist das Vorzeichen-Flag rückgesetzt.

   Beim ADD- und beim SUB-Befehl sind Übertrag-Flag, Hilfs-Übertrag-Flag und Vorzeichen-Flag voneinander unabhängig. Das Überlauf-Flag hängt vom Übertrag-Flag, Vorzeichen-Flag und Subtraktions-Flag ab.

   Beim NEG-Befehl ist das Übertrag-Flag genau dann gelöscht, wenn das Null-Flag gesetzt ist (warum?); Hilfs-Übertrag-Flag und Vorzeichen-Flag sind dagegen voneinander unabhängig. Das Überlauf-Flag hängt vom Vorzeichen-Flag und vom Hilfs-Übertrag-Flag ab.

Kapitel 7.1

1. Der Text lautet: *Gut gemacht!*

Hinter dem Text folgt noch ein Wagenrücklauf (CR) und ein Zeilenvorschub (LF).

2. Die ASCII-Codierung des Textes (mit einem Wagenrücklauf und einem Zeilenwechsel am Ende) lautet:

31H 34H 25H 20H 4DH 65H 68H 72H 77H 65H 72H 74H 73H 74H 65H 75H 65H 72H
20H 76H 6FH 6EH 20H 31H 32H 30H 2EH 2DH 2DH 20H 44H 4DH 2CH 20H 64H 61H
73H 20H 73H 69H 6EH 64H 20H 31H 36H 2EH 38H 30H 20H 44H 4DH 2EH 0DH 0AH

Kapitel 7.2

1. Das Programm lautet beispielsweise

```
SUB     'A'-1           ; Grossbuchstaben auf
                        ; Ordnungszahl abbilden,
                        ; ord('A')=1, ord('B')=2, ...
```

2. Das Programm sieht folgendermaßen aus:

```
ADD     A,'a'-1         ; Ordnungszahl auf
                        ; Kleinbuchstaben abbilden
                        ; ord('a')=1, ord('b')=2, ...
```

Kapitel 8.1

1. Das Programm lautet:

```
LD      A,(7422H)       ; Inhalt der Variablen
SUB     17              ; mit der Adresse 7422H
LD      (7422H),A       ; um 17 vermindern
```

2. Das zugehörige Programm ist:

```
LD      A,(5444H)       ; Inhalt der Variablen
NEG                     ; mit der Adresse 5444H
LD      (5444H),A       ; negieren
```

3. War das sehr schwer?

```
NEG                     ; Reihenfolge der
ADD     A,'A'+'Z'       ; Grossbuchstaben umkehren
```

Kapitel 8.2

1. Das Sechzehnfache erhalten wir mit folgendem Programm:

```
LD      A,C             ; Operand bereitstellen
ADD     A,A             ; Operand verdoppeln
ADD     A,A             ; Operand vervierfachen
ADD     A,A             ; Operand verachtfachen
ADD     A,A             ; Operand versechzehnfachen
```

Beachten Sie dabei, daß der Operand im C-Register steht, also nicht gesichert werden muß; allerdings muß er zur Berechnung dann erst ins A-Register gebracht werden.

2. Ein effizientes Programm lautet:

```
LD      D,A             ; Operand sichern
ADD     A,A             ; Operand verdoppeln
ADD     A,A             ; Operand vervierfachen
ADD     A,A             ; Operand verachtfachen
ADD     A,A             ; Operand versechzehnfachen
SUB     D               ; Operand verfuenfzehnfachen
ADD     A,A             ; Operand verdreissigfachen
```

3. Für <A>=16 und für <A>=17.

4. Das Verfahren nennt man »Ringtausch«:

```
        LD      A,B
        LD      B,D
        LD      D,A
```

Kapitel 9.1

1. Das Programm könnte zum Beispiel lauten:

```
        CP      'a'             ; Inhalt des A-Registers
                                ; mit 'a' vergleichen
        JP      NC,KLEINB       ; springe, wenn Kleinbuchstabe
                                ; im A-Register
        ADD     A,'a' - 'A'     ; wandle Grossbuchstaben
                                ; in Kleinbuchstaben um
KLEINB: NOP                     ; gemeinsame Fortsetzungsstelle
```

2. Die Schwierigkeit besteht darin, das Vorzeichen der Zahl zu testen; hier ein Vorschlag für eine Lösung mit den bisher besprochenen Befehlen (mit Hilfe weiterer Befehle läßt sich das Programm verbessern):

```
        NEG                     ; Inhalt des A-Registers
                                ; negieren, um dabei das
                                ; Vorzeichen des Ergebnisses
                                ; zu erhalten
        JP      P,POSIT         ; springe, wenn Zahl nun positiv
        NEG                     ; berechne Betrag
                                ; der negativen Zahl
POSIT:  NOP                     ; gemeinsame Fortsetzungsstelle
```

3. Als erstes überlegen wir uns, wie der Schlüssel der Codierung dargestellt werden kann. Wir nehmen am besten den Abstand des Buchstabens »A« von seiner Codierung (im Beispiel wäre das also 2). Immer im gleichen Drehsinn rechnen!

   Als nächstes addieren wir diesen Abstand zu unserem Buchstaben; für einige Buchstaben ergibt dies bereits den richtigen Code.

   Haben wir durch die Addition den Bereich der Buchstaben verlassen (im Beispiel passiert dies für die Buchstaben »Y« und »Z«), so müssen wir das »im Kreis wandern« simulieren; wir ziehen 26 (Anzahl der Buchstaben des Alphabets – Umfang unseres Buchstabenkreises) ab.

   Als Programm erhalten wir dann (Schlüssel im B-Register):

```
        ADD     A,B         ; Abstand zu Zeichen addieren
        CP      'Z'+1       ; Feststellen, ob wir den
                            ; Bereich der Buchstaben
                            ; verlassen haben
        JP      C,CAESAR    ; springe, wenn Buchstabe
                            ; im A-Register
        SUB     26          ; zyklisch wieder auf
                            ; Buchstaben abbilden
CAESAR: NOP                 ; CAESAR-Codierung komplett
```

Für die Decodierung brauchen wir obige Operationen nur umzukehren; wir bewegen uns damit um den selben Abstand im Kreis, jedoch diesmal in Gegenrichtung:

```
        SUB     B           ; Abstand von Zeichen
                            ; subtrahieren
        CP      'A'         ; Feststellen, ob wir den
                            ; Bereich der Buchstaben
                            ; verlassen haben
        JP      NC,RASEAC   ; springe, wenn Buchstabe
                            ; im A-Register
        ADD     A,26        ; zyklisch wieder auf
                            ; Buchstaben abbilden
RASEAC: NOP                 ; CAESAR-Decodierung komplett
```

Kapitel 9.2

1. Das Programmstück läßt sich mit unseren Mitteln so programmieren (später werden wir es einfacher lösen):

```
        CP      'a'         ; auf Kleinbuchstaben testen
        JP      NC,KLEINB   ; Kleinbuchstabe
        ADD     A,'a'-'A'   ; Grossbuchstaben in
                            ; Kleinbuchstaben verwandeln
        JP      WEITER      ; weiter an gemeinsamer
                            ; Fortsetzungsstelle
KLEINB: SUB     'a'-'A'     ; Kleinbuchstaben in
                            ; Grossbuchstaben wandeln
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

2. Wir nehmen an, daß die Binärzahl schon im A-Register steht:

```
        CP      10          ; auf Dezimalziffern testen
        JP      NC,BUCHST   ; keine Dezimalziffer
```

```
            ADD     A,'0'       ; binaer-codierte Dezimalziffer
                                ; in ASCII-codierte
                                ; Dezimalziffer wandeln
            JP      WEITER      ; weiter an gemeinsamer
                                ; Fortsetzungsstelle
BUCHST:     ADD     A,'A'-0AH   ; binaer-codierte Hexziffer
                                ; umwandeln
WEITER:     NOP                 ; gemeinsame Fortsetzungsstelle
```

3. Zunächst die zweiseitige Version:

```
            CP      'A'         ; auf Buchstabe testen
            JP      NC,BUCHST   ; Buchstabe
            SUB     '0'         ; ASCII-codierte Dezimalziffer
                                ; in binaer-codierte
                                ; Dezimalziffer wandeln
            JP      WEITER      ; weiter an gemeinsamer
                                ; Fortsetzungsstelle
BUCHST:     SUB     'A'-0AH     ; ASCII-codierte Hexziffer umwandeln
WEITER:     NOP                 ; gemeinsame Fortsetzungsstelle
```

und nun die einseitige:

```
            SUB     '0'         ; Dieser Betrag kann in beiden
                                ; Faellen abgezogen werden
            CP      10          ; auf Dezimalziffer testen
            JP      C,FERTIG    ; Dezimalziffer
            SUB     'A'-0AH-'0' ; Korrektur fuer Buchstabe
FERTIG:     NOP                 ; gemeinsame Fortsetzungsstelle
```

Man erkennt, daß der direkte Weg nicht unbedingt der eleganteste sein muß. In diesem Falle ist die einseitige Version vorzuziehen, da sie weniger Speicherplatz belegt und zudem schneller ist (Ebenso läßt sich auch die vorherige Aufgabe umschreiben, versuche es doch einmal!).

Kapitel 9.3

1. Der ASCII-Code hat 128 Zeichen, jedes Zeichen tritt bei uns also mit der Wahrscheinlichkeit 1:128 auf. Es ergibt sich dann für das ursprüngliche Programm eine durchschnittliche Laufzeit von ca. 34.08 Taktzyklen.

   Nun die Optimierung (sie ergibt sich durch eine geschicktere Bereichsabprüfung):

```
            CP      '9'+1       ; Zunaechst den groessten
                                ; Bereich ausscheiden
```

```
            JP      NC,FRAGEZ       ; Absoluter Sprung,
                                    ; da haeufige Ausfuehrung
            CP      '0'             ; Von den uebrigen 58 Zeichen
                                    ; die Ziffern abtrennen
            JR      NC,WEITER       ; relativer Sprung, da nur in
                                    ; 10/58 der Faelle erfuellt
FRAGEZ:     LD      A,'?'           ; keine Ziffer
WEITER:     NOP                     ; gemeinsame Fortsetzungsstelle
```

Dieses Programm hat nun eine durchschnittliche Laufzeit von nur noch 30.19 Taktzyklen.

2. Folgendes Programm wäre denkbar:

```
            CP      'a'             ; auf kleiner 'a' testen
            JP      C,WEITER        ; kein Kleinbuchstabe, weiter an
                                    ; gemeinsamer Fortsetzungsstelle
            CP      'z'+1           ; auf groesser 'z' testen
            JP      NC,WEITER       ; kein Kleinbuchstabe, weiter an
                                    ; gemeinsamer Fortsetzungsstelle
            SUB     'a'-'A'         ; Kleinbuchstaben in
                                    ; Grossbuchstaben wandeln
WEITER:     NOP                     ; gemeinsame Fortsetzungsstelle
```

3. Hier biete ich zwei Lösungsvorschläge:

Lösung I:

```
            LD      A,B             ; Operanden holen
            LD      B,'?'           ; Ergebnisse fuer den
            LD      E,-1            ; negativen Fall vorbereiten
            CP      16              ; mit 16 vergleichen
            JP      NC,WEITER       ; Inhalt des A-Registers groesser
                                    ; als 15, somit fertig
            LD      E,0             ; gueltige Binaerziffer
            CP      10              ; auf Dezimalziffer vergleichen
            JP      NC,BUCHST       ; Buchstabe
            ADD     A,'0'           ; binaer-codierte Dezimalziffer
                                    ; in ASCII-codierte
                                    ; Dezimalziffer wandeln
            LD      B,A             ; Ergebnis ablegen
            JP      WEITER          ; weiter an gemeinsamer
                                    ; Fortsetzungsstelle
```

```
BUCHST:   ADD     A,'A'-0AH     ; binaer-codierte Hexziffer
                                ; umwandeln
          LD      B,A           ; Ergebnis ablegen
WEITER:   NOP                   ; gemeinsame Fortsetzungsstelle
```

Lösung II:

```
          LD      A,B           ; Operanden holen
          LD      B,'?'         ; Ergebnisse fuer den
          LD      E,-1          ; negativen Fall vorbereiten
          CP      16            ; mit 16 vergleichen
          JP      NC,WEITER     ; Inhalt des A-Registers groesser
                                ; als 15, somit fertig
          LD      E,0           ; gueltige Binaerziffer
          ADD     A,'0'         ; Dieser Betrag muss immmer
                                ; addiert werden
          CP      '9'+1         ; auf Dezimalziffer testen
          JP      C,DEZZIF      ; Dezimalziffer
          ADD     A,'A'-0AH-'0' ; Fehlbetrag ausgleichen
DEZZIF:   LD      B,A           ; Ergebnis ablegen
WEITER:   NOP                   ; gemeinsame Fortsetzungsstelle
```

4. Es müssen nur mehr Bereiche abgefragt werden. Hier ist die Beispiellösung nicht optimal geschrieben, sondern so, daß man noch einmal schön das Prinzip einer Verzweigungskette erkennt:

```
          LD      A,B           ; Operanden holen
          LD      B,'?'         ; Ergebnisse fuer den
          LD      E,-1          ; negativen Fall vorbereiten
          CP      'f'+1         ; Inhalt groesser als 'f' ?
          JP      NC,FERTIG     ; wenn ja, sind wir fertig
          CP      'a'           ; Inhalt kleiner als 'a' ?
          JP      C,KKLEIB      ; kein Kleinbuchstabe
          LD      E,1           ; Ergebnis binaere Zahl
          SUB     'a'-0AH       ; Kleinbuchstaben in binaere
                                ; Zahl umwandeln
          LD      B,A           ; Operanden zurueckspeichern
          JP      FERTIG        ; weiter an gemeinsamer
                                ; Fortsetzungsstelle
KKLEIB:   JP      'F'+1         ; Inhalt größer als 'F' ?
          JP      NC,FERTIG     ; wenn ja, sind wir fertig
          CP      'A'           ; Inhalt kleiner als 'A' ?
          JP      C,KGROSB      ; kein Grossbuchstabe
```

```
        LD      E,1         ; Ergebnis binaere Zahl
        SUB     'A'-0AH     ; Grossbuchstaben in binaere
                            ; Zahl umwandeln
        LD      B,A         ; Operanden zurueckspeichern
        JP      FERTIG      ; weiter an gemeinsamer
                            ; Fortsetzungsstelle
KGROSB: CP      '9'+1       ; Inhalt groesser als '9' ?
        JP      NC,FERTIG   ; wenn ja, sind wir fertig
        CP      '0'         ; Inhalt kleiner als '0' ?
        JP      C,KASCII    ; Zeichen keine ASCII-codierte
                            ; Ziffer
        LD      E,1         ; Ergebnis binaere Zahl
        SUB     '0'         ; ASCII-codierte Dezimalziffer in
                            ; binaer-codierte Dezimalziffer
                            ; umwandeln
        LD      B,A         ; Operanden zurueckspeichern
        JP      FERTIG      ; weiter an gemeinsamer
                            ; Fortsetzungsstelle
KASCII: CP      16          ; mit 16 vergleichen
        JP      NC,FERTIG   ; Inhalt des A-Registers groesser
                            ; als 15, somit fertig
        LD      E,0         ; gueltige Binaerziffer
        CP      10          ; auf Dezimalziffer vergleichen
        JP      NC,BUCHST   ; Buchstabe
        ADD     A,'0'       ; binaer-codierte Dezimalziffer
                            ; in ASCII-codierte
                            ; Dezimalziffer
                            ; umwandeln
        LD      B,A         ; Ergebnis ablegen
        JP      FERTIG      ; weiter an gemeinsamer
                            ; Fortsetzungsstelle
BUCHST: ADD     A,'A'-0AH   ; binaer-codierte Hexziffer
                            ; umwandeln
        LD      B,A         ; Ergebnis ablegen
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Kapitel 9.4

1. Hier ist die Lösung durch die Flußdiagramme und die Programmbeschreibung so eindeutig vorgegeben, daß an dieser Stelle auf ein Listing des Programms verzichtet werden kann. Wer Probleme beim Lösen der Aufgabe hatte, sollte sich noch einmal die anderen Optimierungsstufen genau ansehen.

2. Auch hier soll keine Lösung gegeben werden, um den Leser zur Beschäftigung mit dem Problem zu zwingen.

3. Hier existieren so viele verschiedene Möglichkeiten, daß ein ausformuliertes, womöglich noch optimiertes Programm sicher keinem weiterhilft. Es werden hier lediglich einige Ansätze gezeigt:

   1. Ansatz: Jeder Fall wird vollständig in einem Anlauf bewältigt:

```
wenn    <A>= '1'
dann    wenn <B><> 23
        dann B <-<B>+ 1
        wenn <C><> 0
        dann C <-<C>- 1
sonst   wenn <A>= '2'
        dann    :
                :
                :
```

   2. Ansatz: Fiktiv wird die Bewegung für jeden Fall durchgeführt, anschließend wird dann die Korrektheit geprüft, wonach eventuell die Werte korrigiert werden müssen:

```
wenn    <A>= '1'
dann    B <-<B>+ 1
        C <-<C>- 1
sonst   wenn <A>= '2'
        dann    :
                :
```

   Korrektur: (für Zeile)

```
wenn    <B>= -1
dann    B <- 0
sonst   wenn <B>= 24
        dann B <- 23
```

   Für die Spalte geht es genauso.

Es gibt dann noch die Möglichkeit, die Komponenten getrennt zu behandeln oder die gemeinsamen Befehle hochzuziehen, um das Programm zu optimieren.

4. Es ändert sich im Prinzip an dem vorherigen Programm nichts, es muß nur bei der Fehlerbehandlung der entsprechende andere Wert geladen werden:

Korrektur: (für Zeile)

wenn <B>= -1
dann B <- 23
sonst **wenn** <B>= 24
dann B <- 0

Für die Spalte geht es genauso.

Kapitel 10.1

1. Lösung:

```
        LD      BC,22131        ; BC-Register mit 22131 laden
```

2. Lösung:

```
        LD      D,B             ; Inhalt des BC-Registers in das
        LD      E,C             ; DE-Register bringen
```

3. Lösung:

```
        LD      DE,(2124H)      ; Inhalt des ab der Adresse 2124H
                                ; im Speicher stehenden Worts
                                ; ins DE-Register laden
```

4. Lösung (das HL-Register ergibt einen kürzeren Objekt-Code!):

```
        LD      HL,(4256H)      ; Das ab Adresse 4256H stehende
        LD      (567BH),HL      ; Wort ab Adresse 567BH ablegen
```

Kapitel 10.2

1. Die Deklarationen lauten:

```
BYTE:   DEFB    45H             ; Byte-Variable mit Inhalt 45H
                                ; initialisieren
PLUS:   DEFB    '+'             ; Zeichen-Variable mit Inhalt '+'
                                ; initialisieren
WORT:   DEFW    1700            ; Wort-Variable mit Inhalt 1700
                                ; initialisieren
WUNSCH: DEFM    'Happy New Year!' ; Zeichenketten-Variable
                                ; mit Glueckwunsch initialisieren
```

Dabei belegen:

BYTE	1 Byte
PLUS	1 Byte
WORT	2 Bytes
WUNSCH	15 Bytes

2. Die Deklarationen lauten:

```
VAR1:     DEFS    1          ; Speicherplatz fuer ein Byte
                             ; reservieren
ADR1:     DEFS    2          ; Speicherplatz fuer ein Wort
                             ; reservieren
LGINT:    DEFS    4          ; Speicherplatz fuer 32-Bit-Zahl
                             ; reservieren
STR7:     DEFS    7          ; Speicherplatz fuer Zeichenkette
                             ; mit 7 Zeichen reservieren
PUFFER:   DEFS    128        ; Speicherplatz fuer Puffer
                             ; mit 128 Bytes reservieren
```

Kapitel 10.3

1. Vergleiche hierzu auch MULT10:

```
OP1:      DEFS    1          ; Speicherplatz fuer 8-Bit-Zahl
ERG:      DEFS    2          ; Speicherplatz fuer
                             ; 16-Bit-Ergebnis
MULT12:   LD      A,(OP1)    ; Operanden laden
          LD      L,A        ; Zu einer 16-Bit-
          LD      H,0        ; Groesse erweitern
          LD      D,H        ; Kopie
          LD      E,L        ; erstellen
          ADD     HL,HL      ; Operanden verdoppeln
          ADD     HL,DE      ; Operanden verdreifachen
          ADD     HL,HL      ; Operanden versechsfachen
          ADD     HL,HL      ; Operanden verzwoelffachen
          LD      (ERG),HL   ; Ergebnis abspeichern
```

2. Auch dieses Programm konnte man sofort anhand der gezeigten 32-Bit-Addition herleiten (oder nicht?):

```
OP1:      DEFS    8          ; 64-Bit-Speicherplatz fuer
OP2:      DEFS    8          ; den 1. und 2. Operanden
                             ; reservieren
```

```
ERG:    DEFS    8           ; 64-Bit-Ergebnis reservieren
ADD64:  LD      HL,(OP1)    ; die niederwertigen 16 Bits
        LD      DE,(OP2)    ; der Operanden holen,
        ADD     HL,DE       ; addieren
        LD      (ERG),HL    ; und ablegen
        LD      HL,(OP1+2)  ; die naechsten 16 Bits der
        LD      DE,(OP2+2)  ; Operanden holen,
        ADC     HL,DE       ; mit Uebertrag addieren
        LD      (ERG+2),HL  ; und ablegen
        LD      HL,(OP1+4)  ; die naechsten 16 Bits der
        LD      DE,(OP2+4)  ; Operanden holen,
        ADC     HL,DE       ; mit Uebertrag addieren
        LD      (ERG+4),HL  ; und ablegen
        LD      HL,(OP1+6)  ; die hoechstwertigen 16 Bits
        LD      DE,(OP2+6)  ; der Operanden holen,
        ADC     HL,DE       ; mit Uebertrag addieren
        LD      (ERG+6),HL  ; und ablegen
```

3. Dieses Programm läuft wie bei der Addition ab, wichtig war hier nur, das Übertrag-Flag richtig zu setzen:

```
OP1:    DEFS    8           ; 64-Bit-Speicherplatz fuer
OP2:    DEFS    8           ; den 1. und 2. Operanden
                            ; reservieren
ERG:    DEFS    8           ; 64-Bit-Ergebnis reservieren
SUB64:  SCF                 ; Uebertrag-Flag setzen
        CCF                 ; und loeschen
        LD      HL,(OP1)    ; die niederwertigen 16 Bits
        LD      DE,(OP2)    ; der Operanden holen,
        SBC     HL,DE       ; subtrahieren
        LD      (ERG),HL    ; und ablegen
                :
                :
                :           ; usw. wie oben
```

Kapitel 11.1

1. Die Lösung lautet:

```
        ADD     HL,BC       ; absolute Adresse
                            ; im HL-Register herstellen
        JP      (HL)        ; und diese dann
                            ; indirekt anspringen
```

Kapitel 11.2

1. Die notwendigen Deklarationen seien schon vereinbart:

```
        LD      A,(HL)      ; 1. Operanden holen
        INC     HL          ; Zeiger auf 2. Operanden richten
        ADD     A,(HL)      ; 2. Operanden aufaddieren
        INC     HL          ; Zeiger auf 3. Operanden richten
        ADD     A,(HL)      ; 3. Operanden aufaddieren
        INC     HL          ; Zeiger auf Ergebnis richten
        LD      (HL),A      ; Ergebnis ablegen
```

2. Das Prinzip ist das gleiche wie bei der vorherigen Aufgabe, nur muß man zum Schluß abwärts zählen, um auf das Ergebnis zu zeigen:

```
        LD      A,(HL)      ; 1. Operanden holen
        INC     HL          ; Zeiger auf 2. Operanden richten
        ADD     A,(HL)      ; 2. Operanden aufaddieren
        DEC     HL          ; Zeiger auf
        DEC     HL          ; Ergebnis berechnen
        LD      (HL),A      ; Ergebnis ablegen
```

3. Die Deklarationen seien wie folgt (dabei sollen die Folgen im Speicher verteilt sein):

```
OP1:    DEFS    5           ; 1. Bytefolge
        :
OP2:    DEFS    5           ; 2. Bytefolge
        :
ERG:    DEFS    5           ; Ergebnisfolge
        LD      BC,OP1      ; Zeiger auf die
        LD      HL,OP2      ; Datenbloecke
        LD      DE,ERG      ; initialisieren
        LD      A,(BC)      ; 1. Byte des 1. Feldes laden
        ADD     A,(HL)      ; 1. Byte des 2. Feldes addieren
        LD      (DE),A      ; 1. Ergebnisbyte ablegen
        INC     BC          ; Zeiger auf das
        INC     HL          ; jeweils naechste Byte
        INC     DE          ; richten
        LD      A,(BC)      ; 2. Byte des 1. Feldes laden
        ADD     A,(HL)      ; 2. Byte des 2. Feldes addieren
        LD      (DE),A      ; 2. Ergebnisbyte ablegen
        INC     BC          ; Zeiger auf das
        INC     HL          ; jeweils naechste Byte
```

```
        INC     DE          ; richten
        LD      A,(BC)      ; 3. Byte des 1. Feldes laden
        ADD     A,(HL)      ; 3. Byte des 2. Feldes addieren
        LD      (DE),A      ; 3. Ergebnisbyte ablegen
        INC     BC          ; Zeiger auf das
        INC     HL          ; jeweils naechste Byte
        INC     DE          ; richten
        LD      A,(BC)      ; 4. Byte des 1. Feldes laden
        ADD     A,(HL)      ; 4. Byte des 2. Feldes addieren
        LD      (DE),A      ; 4. Ergebnisbyte ablegen
        INC     BC          ; Zeiger auf das
        INC     HL          ; jeweils naechste Byte
        INC     DE          ; richten
        LD      A,(BC)      ; 5. Byte des 1. Feldes laden
        ADD     A,(HL)      ; 5. Byte des 2. Feldes addieren
        LD      (DE),A      ; 5. Ergebnisbyte ablegen
```

Später werden wir solche Probleme natürlich mittels einer Schleife allgemeiner formulieren.

4. Als Adreßregister benutzen wir das HL-Register. Das Programmstück lautet:

```
        LD      BC,30       ; Distanz ins BC-Register laden
        LD      E,(HL)      ; 1. Byte holen
        INC     HL          ; auf 2. Byte zeigen
        LD      D,(HL)      ; 2. Byte holen
        ADD     HL,BC       ; Zieladresse des 2. Bytes
                            ; erzeugen
        LD      (HL),D      ; 2. Byte ablegen
        DEC     HL          ; Auf Zieladresse des 1. Bytes
                            ; zeigen
        LD      (HL),E      ; 1. Byte ablegen
```

Kapitel 12.1

1. Ob die Zahl ungerade ist, erkennt man an dem letzten Bit:

```
        BIT     0,E         ; letztes Bit testen
        JP      Z,WEITER    ; Zahl gerade
        DEC     E           ; Zahl um eins vermindern
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

2. Hier ist das 5. Bit entscheidend:

```
        LDD     ,0          ; Annahme, der Buchstabe sei klein
        BIT     5,H         ; 5. Bit testen
        JP      NZ,WEITER   ; Kleinbuchstabe
        LD      D,-1        ; Grossbuchstaben signalisieren
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

3. Dies war ja eigentlich nicht schwer:

```
        BIT     7,(HL)      ; Testet das Vorzeichen einer durch das
                            ; HL-Register adressierten Zahl
        JP      Z,POSIT     ; verzweigt, falls positiv
        :
        :
POSIT:  :
        :
```

Kapitel 12.2

1. Vergleiche hierzu auch die Aufgabe 9.2.1.; das Prinzip der Aufgabe bleibt hier gleich:

```
        BIT     5,(HL)      ; Teste entscheidendes Bit
        JP      NZ,KLEINB   ; Kleinbuchstabe
        SET     5,(HL)      ; Grossbuchstaben in
                            ; Kleinbuchstaben umwandeln
        JP      WEITER      ; weiter an gemeinsamer
                            ; Fortsetzungsstelle
KLEINB: RES     5,(HL)      ; Kleinbuchstaben in
                            ; Grossbuchstaben umwandeln
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

2. Das Programm lautet:

```
        BIT     7,B         ; Vorzeichen testen
        JP      NZ,NEGAT    ; negativ
        SET     7,B         ; negatives Vorzeichen setzen
        JP      WEITER      ; weiter an gemeinsamer
                            ; Fortsetzungsstelle
NEGAT:  RES7,B              ; positives Vorzeichen setzen
WEITER: NOP                 ; gemeinsame Fortsetzungsstelle
```

Kapitel 12.3

1. Das Übertrag-Flag soll beispielsweise im Bit 1 des D-Registers gespeichert werden:

```
            :
            SET     1,D         ; gesetztes Flag annehmen
            JP      C,WEITER    ; Uebertrag-Flag gesetzt
            RES     1,D         ; als rueckgesetzt markieren
WEITER:     NOP                 ; gemeinsame Fortsetzungsstelle
```

2. Der Ablauf ist wie bei der vorherigen Aufgabe:

```
FLAG:       DEF     S1          ; Platz für die Speicherung
            :
            LDH     L,FLAG      ; Zeiger auf die Speicheradresse laden
            SET     6,(HL)      ; gesetztes Flag annehmen
            JP      Z,WEITER    ; Null-Flag gesetzt
            RES     6,(HL)      ; als rueckgesetzt markieren
WEITER:     NOP                 ; gemeinsame Fortsetzungsstelle
```

Kapitel 12.4

1. Für diese Umwandlung muß man nur den oberen Nibble ausblenden:

```
            AND     00001111B   ; loescht den oberen Nibble
```

2. In dieser Zahldarstellung muß nur das oberste Bit gelöscht werden:

```
            AND     01111111B   ; loescht Bit 7
```

3. Ohne Bereichsüberprüfung genügt es, das Bit 5 zu setzen:

```
            OR      00100000B   ; Setzt das Bit 5
```

4. Und nun endlich das versprochene kurze Programm (vgl. Aufgabe 9.2.1 und 12.2.1):

```
            XOR     00100000B   ; invertiert das Bit 5,
                                ; das heisst vertauscht
                                ; Gross- und Kleinbuchstaben
```

5. Auch hier erfolgt wieder keine Bereichsüberprüfung:

```
            XOR     30H         ; vertauscht ASCII- und
                                ; Binaer-Codierung
                                ; von Dezimalziffern
```

6. Die umzuwandelnde Zahl stehe bereits im A-Register. Der Typ der umzuwandelnden Zahl wird im B-Register durch die Funktionsnummern 0 (Vorzeichen/Betrag-Darstellung), 1 (1-Komplement), 2 (2-Komplement) festgelegt. Die Ergebnisse finden sich in den entsprechenden Speicherzellen. Man erhält außerdem im A-Register den Wert 0, falls die Umwandlung korrekt durchgeführt wurde, sonst FFH.

```
VZB:    DEF     S1          ; Speicherplatz fuer
                            ; Vorzeichen/Betrag-Darstellung
KPL1:   DEF     S1          ; Speicherplatz für
                            ; 1-Komplement-Darstellung
KPL2:   DEF     S1          ; Speicherplatz für
                            ; 2-Komplement-Darstellung
        OR      A           ; Trick, um das Vorzeichen
                            ; zu bekommen
        JP      P,POSIT     ; Zahl positiv,
                            ; dann alle Darstellungen gleich
        DEC     B           ; auf Komplement-Darstellung
                            ; testen
        JP      P,KOMP      ; Komplement-Darstellung
        LD      (VZB),A     ; Zahl ist in
                            ; Vorzeichen/Betrag-Darstellung
        AND     7FH         ; Vorzeichen positiv machen
        CPL                 ; erzeugt das 1-Komplement
        LD      (KPL1),A    ; 1-Komplement ablegen
        INC     A           ; erzeugt das 2-Komplement
        LD      (KPL2),A    ; 2-Komplement ablegen
        JP      OK          ; Aufgabe erledigt
                            ; mit korrekter Bearbeitung
KOMP:   JP      NZ,KOMP2    ; 2-Komplement-Darstellung
        LD      (KPL1),A    ; Zahl ist 1-Komplement
        INC     A           ; 2-Komplement erzeugen
        LD      (KPL2),A    ; 2-Komplement ablegen
        NEG                 ; in positive Zahl umwandeln
        OR      10000000B   ; negatives Vorzeichen setzen
        LD      (VZB),A     ; Vorzeichen/Betrag ablegen
        JP      OK          ; Aufgabe erledigt
                            ; mit korrekter Bearbeitung
KOMP2:  LD      (KPL2),A    ; Zahl ist im 2-Komplement
        DEC     A           ; 1-Komplement erzeugen
        JP      P,FEHLER    ; Zahl war -128, diese
                            ; ist in den beiden anderen
                            ; Formaten nicht darstellbar
        LD      (KPL1),A    ; 1-Komplement abspeichern
```

```
            CPL                     ; in positive Zahl verwandeln
            OR      10000000B       ; negative Vorzeichen setzen
            LD      (VZB),A         ; Vorzeichen/Betrag ablegen
            JP      OK              ; Aufgabe erledigt
                                    ; mit korrekter Bearbeitung
POSIT:      LD      (VZB),A         ; positive
            LD      (KPL1),A        ; Darstellungen
            LD      (KPL2),A        ; ablegen
OK:         LD      A,0             ; 0 = korrekte Bearbeitung
            JP      FERTIG          ; Umwandlung beendet
FEHLER:     LD      A,0FFH          ; FFH = Fehler aufgetreten
FERTIG:     NOP                     ; gemeinsame Fortsetzungsstelle
```

7. Man kann das gleiche Programm wie in Aufgabe 6 verwenden. Die Anpassung auf 16 Bit ist nicht allzuschwer, da auf der Zahl ja nur einfache Operationen (INC, DEC, CPL, NEG) durchgeführt wurden. Diese müssen also für 16 Bit (zum Beispiel im HL-Register) nachgebildet werden:

```
CPLHL:      LD      A,H             ; hoeherwertiges Byte laden
            CPL                     ; komplementieren
            LD      H,A             ; und zurueckspeichern
            LD      A,L             ; niederwertiges Byte laden
            CPL                     ; komplementieren
            LD      L,A             ; und zurueckspeichern
```

NEG wird auf ähnliche Weise behandelt; hier sind die beiden Bytes jedoch nicht unabhängig voneinander.

Kapitel 12.5

1. Eine mögliche Lösung wäre:

```
            LD      B,A             ; Byte sichern
            AND     0FH             ; oberen Nibble ausblenden
            ADD     A,'0'           ; Umwandlung fuer Dezimalziffer
            CP      '9'+1           ; auf Dezimalziffer testen
            JP      C,ZIFF1         ; Dezimalziffer
            ADD     A,'A'-'0'-0AH   ; Korrektur fuer Hex-Buchstabe
ZIFF1:      LD      C,A             ; niederwertiges Zeichen
                                    ; abspeichern
            LD      A,B             ; Byte wieder holen
            SRL     A               ; viermal nach
```

```
        SRL     A           ; rechts schieben, um
        SRL     A           ; die oberen vier Bits
        SRL     A           ; zu isolieren
        ADD     A,'0'       ; Umwandlung fuer Dezimalziffer
        CP      '9'+1       ; auf Dezimalziffer testen
        JP      C,ZIFF2     ; Dezimalziffer
        ADD     A,'A'-'0'-0AH ; Korrektur fuer Hex-Buchstabe
ZIFF2:  LD      B,A         ; hoeherwertiges Zeichen
                            ; abspeichern
```

2. Diese Aufgabe ist vom Ablauf kaum von der vorherigen verschieden:

```
        LD      A,D         ; hoeherwertiges Zeichen holen
        SUB     A,'0'       ; Umwandlung fuer Dezimalziffer
        CP      10          ; auf Dezimalziffer testen
        JP      C,ZIFF1     ; Dezimalziffer
        SUB     A,'A'-'0'-0AH ; Korrektur fuer Hex-Buchstabe
ZIFF1:  SLA     A           ; viermal nach
        SLA     A           ; links schieben, um
        SLA     A           ; die unteren vier Bits in
        SLA     A           ; den oberen Nibble zu bringen
        LD      D,A         ; hoeherwertigen Nibble sichern
        LD      A,E         ; niederwertiges Zeichen holen
        SUB     A,'0'       ; Umwandlung fuer Dezimalziffer
        CP      10          ; auf Dezimalziffer testen
        JP      C,ZIFF2     ; Dezimalziffer
        SUB     A,'A'-'0'-0AH ; Korrektur fuer Hex-Buchstabe
ZIFF2:  OR      D           ; niederwertigen Nibble mit
                            ; hoeherwertigem verknuepfen
```

3. Drei kurze Programme:

```
        SRA     B           ; arithmetische
        RR      C           ; Rechtsverschiebung
        RR      D
        RR      E
        SRL     B           ; logische
        RR      C           ; Rechtsverschiebung
        RR      D
        RR      E
        SLA     E           ; arithmetische
        RL      D           ; Linksverschiebung
        RL      C
        RL      B
```

4. In diesem Fall ist es sinnvoll, die Nibble-Rotierbefehle einzusetzen:

```
BYTE:   DEFS    1           ; Byte im Speicher
        :
        XOR     A           ; Akkumulator loeschen
        LD      HL,BYTE     ; Zeiger auf das Byte laden
        RR      D           ; niederwertigen Nibble holen
        ADD     A,'0'       ; Umwandlung fuer Dezimalziffer
        CP      '9'+1       ; auf Dezimalziffer testen
        JP      C,ZIFF1     ; Dezimalziffer
        ADD     A,'A'-'0'-0AH ; Korrektur fuer Hex-Buchstabe
ZIFF1:  LD      C,A         ; niederwertiges Zeichen ablegen
        XOR     A           ; Akkumulator loeschen
        RR      D           ; hoeherwertigen Nibble holen
        ADD     A,'0'       ; Umwandlung fuer Dezimalziffer
        CP      '9'+1       ; auf Dezimalziffer testen
        JP      C,ZIFF2     ; Dezimalziffer
        ADD     A,'A'-'0'-0AH ; Korrektur fuer Hex-Buchstabe
ZIFF2:  LD      B,A         ; hoeherwertiges Zeichen ablegen
```

Die 2. Aufgabe geht im Prinzip genauso. Deshalb soll an dieser Stelle auf eine Lösung verzichtet werden.

Kapitel 13.1

1. Lösung:

```
        LD      A,00000001B ; Maske für Bit 0
        LD      B,C         ; Zaehler holen
        INC     B           ; Schleife abweisend
        JP      TEST        ; machen
SCHIEB: SLA     A           ; Maske um 1 Bit
                            ; nach links schieben
TEST:   DJNZ    SCHIEB      ; n-mal schieben
```

2. Der Programmaufbau ist wie bei der vorherigen Aufgabe, lediglich der Schleifenkörper wird umfangreicher:

```
        LD      B,A         ; Zaehler laden
        INC     B           ; Schleife abweisend
        JP      TEST        ; machen
ROTIER: SRL     D           ; D-Register logisch rechts
```

```
                                        ; schieben, das heisst
                                        ; Bit 7 von D ist nun 0 !
        RR      E                       ; altes Bit 0 von D in E
                                        ; hineinrotieren
        JP      NC,TEST                 ; kein Übertrag, Bit 0 von E
                                        ; war 0, somit fertig
        SET     7,D                     ; sonst muss noch das Bit 7 in D
                                        ; gesetzt werden,
                                        ; da wir ja rotieren wollen
TEST:   DJNZ    ROTIER                  ; n-mal rotieren
```

3. Gegenüber der Multiplikation vorzeichenloser ganzer Zahlen sind folgende Änderungen durchzuführen:
- Ist der Multiplikator C negativ, so negieren wir Multiplikator und Multiplikand; dadurch bleibt das Vorzeichen des Ergebnisses erhalten, der Multiplikator wird aber positiv.
- Das D-Register erhält den Wert FFH, falls in E eine negative Zahl steht, sonst den Wert 00H (Erweiterung einer vorzeichenbehafteten ganzen Zahl in 2-Komplement-Darstellung von 8 Bit auf 16 Bit).

```
        LD      HL,0                    ; Akkumulator loeschen
MNDPOS: LD      A,C                     ; Multiplikator holen
        OR      A                       ; und Vorzeichen testen
        JP      P,MTRPOS                ; Multiplikator positiv,
                                        ; keine Korrektur noetig
        NEG                             ; Multiplikator negieren
        LD      C,A                     ; und zurueckschreiben
        LD      A,E                     ; Multiplikand holen
        NEG                             ; Multiplikand negieren
        LD      E,A                     ; und zurueckschreiben
MTRPOS: LD      D,H                     ; Multiplikand zu 16 Bit
                                        ; Groesse erweitern, zunaechst
                                        ; positiven Multiplikanden
                                        ; annehmen
        BIT     7,E                     ; Vorzeichen des Multiplikanden
                                        ; testen
        JP      Z,MNDPOS                ; positiver Multiplikand
        DEC     D                       ; Multiplikand korrekt zu Wort
                                        ; erweitern
MNDPOS: LD      B,8                     ; Schleifenzaehler mit Laenge
                                        ; des Multiplikators (in Bits)
                                        ; besetzen
MULTI:  ADD     HL,HL                   ; Akkumulator verdoppeln
        RLC     C                       ; hoechstes Bit des
```

```
                                        ; Multiplikators ins Uebertrag-
                                        ; Flag bringen, gleichzeitig
                                        ; Multiplikator links-rotieren
        JP      NC,NULL                 ; keine Addition erforderlich,
                                        ; wenn anstehendes Bit des
                                        ; Multiplikators 0 ist
        ADD     HL,DE                   ; Multiplikand zu Akkumulator
                                        ; addieren
NULL:   DJNZ    MULTI                   ; naechstes Bit des
                                        ; Multiplikators verarbeiten
```

Kapitel 13.2

1. Lösung:

```
        LD      HL,0                    ; Akkumulator loeschen
        LD      A,4                     ; Zaehler laden
        LD      D,0                     ; fuer 16-Bit-Summation
                                        ; vorbereiten
ADD:    LD      E,A                     ; aktuellen Summanden laden
        ADD     HL,DE                   ; aufaddieren
        INC     A                       ; Zaehler erhoehen
        CP      18                      ; mit Endwert vergleichen
        JP      C,ADD                   ; Schleifenkörper wiederholen,
                                        ; falls <A> <= 17
```

2. Lösung:

```
        LD      HL,0                    ; Akkumulator loeschen
        LD      A,1                     ; ersten ungeraden Zaehler laden
        LD      D,0                     ; fuer 16-Bit-Summation
                                        ; vorbereiten
ADD:    LD      E,A                     ; aktuellen Summanden laden
        ADD     HL,DE                   ; aufaddieren
        ADD     A,2                     ; Zähler um 2 erhöhen
                                        ; (nur ungerade Zahlen !)
        CP      31                      ; mit Endwert vergleichen
        JP      C,ADD                   ; Schleifenkörper wiederholen,
                                        ; falls <A> <= 30
```

3. Lösung:

```
        LD      HL,51A7H        ; Zeiger auf erste betroffene
                                ; Speicherzelle
        LD      BC,5242H-51A6H  ; Laenge des Speicherbereichs
FUELL:  LD      (HL),0FFH       ; Speicherzelle fuellen
        INC     HL
        INC     HL              ; Zeiger auf naechste betroffene
                                ; Speicherzelle
        DEC     BC
        DEC     BC              ; Schrittweite zu Zaehler addieren
        LD      A,B             ; Test auf <BC> = 0
                                ; vorbereiten
        OR      C               ; JPNZ,FUELL; Zaehler ungleich 0,
                                ; zum Schleifenanfang springen
```

4. Wir benötigen hier zwei Zählregister und einen Akkumulator. Der Einfachheit halber wollen wir annehmen, daß der Wert von N kleiner als 255 ist:

```
N:      DEFS    1               ; Platz fuer die Zaehlgroesse
        :
        LD      HL,0            ; Akkumulator loeschen
        LD      A,N             ; Zaehler der aeusseren
                                ; Schleife in Akku
        LD      D,0             ; fuer 16-Bit-Summation
                                ; vorbereiten
AUSSEN: LD      B,A             ; innere Schleife mittels des
                                ; automatischen
                                ; Schleifenbefehls abwickeln
INNEN:  LD      E,B             ; Summand laden
        ADD     HL,DE           ; aufaddieren
        DJNZ    INNEN           ; innere Schleifenkontrolle
        DEC     A               ; aeusseren Zaehler erniedrigen
        JP      NZ,AUSSEN       ; Schleife wiederholen,
                                ; falls nicht fertig
```

5. Beim Abwärtszählen der Speicherzelle muß der Übertrag vom Lowbyte zu Highbyte manuell korrigiert werden:

```
ZAEHLW: DEF     W5000H-3000H    ; Zaehlgroesse initialisieren
        :
        LD      A,0             ; einzutragenden Wert
        LD      DE,3000H        ; Anfangsadresse laden
```

```
        LD      HL,ZAEHLW   ; Zeiger auf unseren Zaehler
FUELL:  LD      (DE),A      ; Speicherzelle mit dem
                            ; im A-Register gespeicherten
                            ; Wert ueberschreiben
        INC     DE          ; Zeiger auf naechste
                            ; Speicherstelle
        DEC     (HL)        ; niederwertiges Byte des
                            ; Zaehlers erniedrigen
        JP      NZ,FUELL    ; Falls < >0, weitermachen
        INC     HL          ; sonst muß das hoeherwertige
        DEC     (HL)        ; Byte korrigiert werden
        DEC     HL          ; Zeiger wieder auf
                            ; niederwertiges Byte stellen
                            ; (dieser Befehl ändert
                            ; die Flags nicht!)
        JP      NZ,FUELL    ; hoeherwertiges Byte ist
                            ; noch >0, weitermachen
```

Kapitel 13.3

1. Das erste Beispielprogramm in diesem Unterkapitel kann schon als Lösung genommen werden. Es genügt dann, den Zähler immer nur um 1 zu vermindern.

2. Die Lösung in der aufsteigenden Variante: (n stehe im C-Reg.)

```
        LD      HL,0        ; Akkumulator loeschen
        LD      D,H         ; D <— 0
        LD      A,3         ; Startwert der Schleife
ADD:    CP      C           ; Auf Zaehler >n testen
        JP      C,KOERP     ; Zaehler < n ,
                            ; Schleifenkoerper ausfuehren
        JP      NZ,FERTIG   ; Zaehler >n , Schleife beenden
KOERP:  LD      E,A         ; Summand ins DE-Register bringen
        ADD     HL,DE       ; und aufsummieren
        ADD     A,3         ; Schrittweite zu Zaehler addieren
        JP      NC,ADD      ; Zaehler < 256 ,
                            ; Schleife fortsetzen
FERTIG: NOP                 ; Fortsetzungspunkt
```

Kapitel 13.4

1. Diese Schleife läßt sich recht einfach gestalten: (Die Anfangsadresse des Speicherbereichs sei im HL-Register)

```
        LD      A,' '       ; Vergleichswert laden
SUCHE:  INC     HL          ; Zeiger erhoehen
        CP      (HL)        ; Vergleiche den Inhalt der
                            ; Zelle mit
                            ; dem gesuchten Zeichen
        JP      NZ,SUCHE    ; falls nicht gefunden,
                            ; weitersuchen
```

Bemerkung: Das obige Programm läuft allerdings endlos, falls im gesamten Speicher kein Leerzeichen vorhanden ist, da wir keine obere Schranke angegeben haben.

2. Die Lösung läßt sich mühelos aus dem im Text gezeigten Beispiel der Fibonacci-Zahlen herleiten. Anstelle der Berechnung einer neuen Fibonacci-Zahl muß nun eine Multiplikation des Vergleichswertes mit 3 erfolgen. Auf ein Programm soll deshalb hier verzichtet werden.

Kapitel 13.5

1. Das Null-Flag ist genau dann gesetzt, wenn die Maximalzahl der Schleifendurchläufe ausgeführt wurde. Es ist nur dann gelöscht, wenn Bit 0 des HL-Registers gesetzt ist.

2. Lösung:

```
        LD      B,8         ; maximale Anzahl
                            ; der Verschiebungen + 1
                            ; fuer abweisende Schleife
        JP      EINSPR      ; in Schleife einspringen
SCHIEB: SLA     A           ; A-Register nach links schieben
EINSPR: CPC                 ; mit C-Register vergleichen
        JP      Z,WEITER    ; <A>=<C>, weitermachen
        JP      NC,FERTIG   ; <A> > <C>, Schleife abbrechen
WEITER: DJNZ    SCHIEB      ; Schleife wiederholen,
                            ; falls noch nicht
                            ; Maximalzahl der Wiederholungen
FERTIG:NOP                  ; Fortsetzungspunkt
```

Beachte hierbei, wie die Bedingung »größer« realisiert wurde. (Haben Sie an den Fall '=' gedacht?)

3. Diese Schleife läßt sich recht einfach gestalten: (Die Endadresse des Speicherbereichs sei im HL-Register)

```
        LD      A,'*'       ; Vergleichswert laden
SUCHE:  CP      (HL)        ; Vergleiche den Inhalt
                            ; der Zelle mit
                            ; dem gesuchten Zeichen
        JP      Z,FERTIG    ; Falls gefunden, Schleife beenden
        DEC     HL          ; Zeiger erniedrigen
        JP      SUCHE       ; Weitersuchen
FERTIG: NOP                 ; Fortsetzungspunkt
```

Bemerkung: Das obige Programm läuft allerdings endlos, falls im gesamten Speicher kein Stern vorhanden ist, da wir keine Beschränkung angegeben haben.

Kapitel 13.7

1. Eine Möglichkeit wäre, die Endadresse+1 im DE-Register zu halten, dann lautet die kritische Stelle:

```
        :
        OR      A
        SBC     HL,DE
        JP      NC,FERTIG   ; nächste Adresse > Endadresse
VERGL:  LD      H,B
        :
```

Eine weitere Möglichkeit besteht darin, das Übertrag-Flag vor der Subtraktion zu setzen:

```
        :
        SCF
        SBC     HL,DE
        JP      NC,FERTIG   ; nächste Adresse > Endadresse
VERGL:  LD      H,B
        :
```

2. Für die Lösung muß man nur das in Kapitel 13.1 gezeigte Multiplikationsprogramm modifizieren:

```
        XOR     A           ; Akkumulator loeschen
        LD      B,8         ; Schleifenzaehler mit
                            ; Multiplikatorlaenge
                            ; initialisieren
```

```
MULT:   ADD     A,A         ; Akkumulator verdoppeln
        JP      C,FEHLER    ; Uebertrag bedeutet:
                            ; Ergebnis nicht mit
                            ; 8 Bits darstellbar
        RLC     C           ; hoechstes Bit
                            ; des Multiplikators in
                            ; Uebertrag-Flag bringen
        JP      NC,NADD     ; keine Addition erforderlich,
                            ; falls Uebertragsbit = 0
        ADD     A,E         ; Sonst Multiplikand aufaddieren
        JP      C,FEHLER    ; Uebertrag bedeutet:
                            ; Ergebnis nicht mit
                            ; 8 Bits darstellbar
NADD:   DJNZ    MULT        ; naechstes Bit des
                            ; Multiplikators verarbeiten
```

3. Lösung: (i sei im B-Register)

```
        LD      HL,1        ; Akkumulator mit 1 vorbesetzen
        INC     B           ; Schleife abweisend
        JP      EINSPR      ; machen
POT:    ADD     HL,HL       ; Akkumulator mit 2 multiplizieren
        JP      C,FEHLER    ; Falls Uebertragsbit gesetzt,
                            ; ist das Ergebnis nicht
                            ; mit 16 Bit darstellbar
EINSPR: DJNZ    POT         ; Weiter potenzieren, falls noetig
```

Kapitel 14.1

1. Lösung:

```
IU      EQU     1               ; kleinster Index
IO      EQU     10              ; groesster Index
LAENGE  EQU     2               ; Wortfeld
FELD:   DEFS    (IO-IU+1)*LAENGE    ; unitialisiertes Feld
                                ; reservieren
IU      EQU     0               ; kleinster Index
IO      EQU     15              ; groesster Index
LAENGE  EQU     1               ; Bytefeld
FELD:   DEFS    (IO-IU+1)*LAENGE    ; unitialisiertes Feld
                                ; reservieren
IU      EQU     -5              ; kleinster Index
```

```
IO          EQU         5               ; groesster Index
LAENGE      EQU         4               ; Feld mit 4-Byte-Einträgen
FELD:       DEFS        (IO-IU+1)*LAENGE    ; unitialisiertes Feld
                                        ; reservieren
FELD:       DEFS        4               ; Feld mit 25 Bits reservieren
FELD:       DEFS        7               ; Feld mit 13 Nibbles reservieren
FELD:       DEFS        8               ; Feld mit 32*2 Bits reservieren
```

2. Die Vereinbarungen lauten:

```
BYFELD:     DEFB        8               ; 1. Feldelement
            DEFB        -13             ; 2. Feldelement
            DEFB        17              ; 3. Feldelement
            DEFB        99              ; 4. Feldelement
            DEFB        -121            ; 5. Feldelement
            DEFB        44              ; 6. Feldelement
WOFELD:     DEFW        12380           ; 1. Feldelement
            DEFW        16421           ; 2. Feldelement
            DEFW        246             ; 3. Feldelement
            DEFW        -13131          ; 4. Feldelement
BIFELD:     DEFB        11011000B       ; Bit 7 - Bit 0
            DEFB        00001011B       ; Bit 12 - Bit 8
NIFELD:     DEFB        0A3H            ; Nibble 1 und Nibble 0
            DEFB        07H             ; Nibble 3 und Nibble 2
            DEFB        0FBH            ; Nibble 5 und Nibble 4
            DEFB        041H            ; Nibble 7 und Nibble 6
            DEFB        0DH             ; Nibble 8
```

3. Die Lösung für die Wortmatrix sieht wie das im Text gebrachte Beispiel der Bytematrix aus. DEFB muß dort durch DEFW ersetzt werden.

   Lösung für die Bitmatrix:

```
BITZEI:     DEFB        01110101B       ; initialisierte Bitmatrix
            DEFB        00000001B       ; vereinbaren (zeilenweise)
BITSPA:     DEFB        11010101B       ; initialisierte Bitmatrix
            DEFB        00000001B       ; vereinbaren (spaltenweise)
```

Kapitel 14.2

1. Die 2-Komplement-Darstellung ändert an der Routine ADRESS nichts, da der SUB-Befehl auch 2-Komplement-Arithmetik richtig ausführt. Der Abstand $i-i_u$ ist (bei korrektem i) nie-

mals negativ, aus diesem Grund bleibt die relative Adressierung ebenfalls korrekt (man kann sich das leicht anhand eines Zahlenbeispiels klarmachen).

2. Man kann hier das Schema der 8-Bit-Routine verwenden, es müssen aber die Registerbelegungen geändert werden:

HL = Index i
BC = kleinster Index i_u
A = Länge eines Feldelements
DE = Anfangsadresse des Feldes

```
BASIS:    DEFS    2           ; Hilfsspeicherplatz
ADRESS:   OR      A           ; Uebertrag-Flag loeschen
          SBC     HL,BC       ; relativen Index berechnen
          EX      DE,HL       ; Register tauschen,
                              ; Relativadresse als
                              ; Multiplikanden nehmen
          LD      (BASIS),HL  ; Basisadresse temporaer sichern
                              ; Multiplikation l*(i-iu) durchfuehren:
          LD      HL,0        ; Akkumulator loeschen
          LD      B,8         ; Schleifenzaehler laden
MULT:     ADD     HL,HL       ; Akkumulator verdoppeln
          RLCA                ; hoechstes Bit des
                              ; Multiplikators ins
                              ; Uebertrag-Flag bringen
          JP      NC,NULL     ; keine Addition noetig, wenn
                              : anstehendes Bit
                              ; des Multiplikators 0 ist
          ADD     HL,DE       ; Multiplikand zu Akkumulator
                              ; addieren
NULL:     DJNZ    MULT        ; naechstes Bit verarbeiten
                              ; Anfangsadresse berechnen
          LD      DE,(BASIS)  ; Anfangsadresse des Felds laden
          ADD     HL,DE       ; Adresse des Elements ausrechnen
```

3. Ich führe hier nicht mehr die ganze Routine auf, sondern nur den davorstehenden Kopf. Für die Indexgrenzenüberwachung verwende man die Routine DYNKON (wie im Text gezeigt).

```
                              ;Deskriptor eines Felds
FELD:
MININD:   DEFB    ...         ; kleinster Index
MAXIND:   DEFB    ...         ; groesster Index
```

```
LAENGE:  DEFB     ...          ; Laenge eines Feldelements
FELADR:  DEFW     ...          ; Adresse des ersten Feldelements
                               ; Routine zur Parameteruebernahme:
                               ; HL zeige auf die Deskriptoradresse,
                               ; im A-Register stehe der Index
GETPAR:  LD       C,(HL)       ; kleinster Index ins C-Register
         LD       D,(HL)       ; groesster Index ins D-Register
         NC       HL
         LD       E,(HL)       ; Laenge laden
         INC      HL
         LD       B,(HL)       ; niederwertiges Byte
                               ; der Adresse laden
         INC      HL
         LD       H,(HL)       ; hoeherwertiges Byte
                               ; der Adresse laden
         LD       L,B
DYNKON:  :
         :
         :
```

4. Das Deskriptorformat kann zum Beispiel folgende Form haben:

```
FELD:
IU1:     DEFB     1            ; kleinster Zeilenindex
IO1:     DEFB     3            ; groesster Zeilenindex
IU2:     DEFB     1            ; kleinster Spaltenindex
IO2:     DEFB     4            ; groesster Spaltenindex
I1:      DEFB     ...          ; Zeile
I2:      DEFB     ...          ; Spalte
LAENGE:  DEFB     2            ; Laenge eines Feldelements
ADR:     DEFW     ...          ; Adresse des Feldes
```

Eine schöne Lösung mit Hilfe von Indexregistern (siehe Kapitel »Verbunde«) soll hier skizziert werden:

```
ADRES2:  LD       IX,FELD      ; Deskriptorbasis laden
                               ; Index ausrechnen, wie im Text
                               ; beschrieben
         LD       A,(IX+4)     ; I1 laden
         SUB      (IX+0)       ; I1-IU1
         LD       E,A          ; als Multiplikand verwenden
         LD       D,0
         LD       A,(IX+3)     ; IO2 laden
```

```
        SUB     (IX+2)      ; IU2 abziehen
        ADD     1           ; und relative Distanz berechnen
MULTI:  :
        :                   ; multiplizieren, wie gewohnt
        :
NULL:   DJNZ    MULTI
        LD      D,0         ; Spaltenindex laden
        LD      E,(IX+5)
        ADD     HL,DE       ; und aufaddieren
        LD      E,(IX+2)    ; Anfangsspaltenindex laden
        OR      A           ; Uebertrag-Flag loeschen
        SBC     HL,DE       ; abziehen
                            ; nun Datenformat fuer die Routine
                            ; ADRESS herstellen
                            ; (der Index steht schon im
                            ; HL-Register)
        LD      BC,0        ; kleinster Index ist 0
        LD      E,(IX+7)
        LD      D,(IX+8)    ; Feldadresse in DE
        LD      A,(IX+6)    ; Elementlaenge in Akku
```

5. In diesem Fall muß in die Adreßberechnung noch die Länge l mit einbezogen werden. Dies geschieht prinzipiell wie bei der Routine ADRESS (zumindest an der gleichen Stelle im Algorithmus). Oder man führt noch eine weitere Pseudo-Dimension der Länge l ein. Die Ausführung dieser Ideen sei hier dem Leser überlassen.

Kapitel 14.3

1. Lösung (das HL-Register zeige auf das letzte Feldelement):

```
        LD      B,255       ; Laenge des Feldes - 1 laden
INIT:   LD      (HL),B      ; Wert initialisieren
        DEC     HL          ; Zeiger auf naechstes
                            ; Feldelement
        DJNZ    INIT        ; alle Elemente mit Ausnahme
                            ; des ersten bearbeiten
        LD      (HL),B      ; erstes Element initialisieren
```

2. DE zeige auf das erste Feld, HL auf das zweite; B enthalte die Anzahl der Feldelemente:

```
VERGL:  LD      A,(DE)      ; niederwertige Bytes
        CP      (HL)        ; vergleichen
```

```
            JP      NZ,FERTIG   ; Elemente verschieden
            INC     HL          ; auf hoeherwertige Bytes
            INC     DE          ; zeigen
            LD      A,(DE)      ; hoeherwertige Bytes
            CP      (HL)        ; vergleichen
            JP      Z,WEITER    ; Elemente gleich
            DEC     HL          ; auf Elemente
            DEC     DE          ; zeigen
            JP      FERTIG      ; Abbruch der Suche
WEITER:     INC     HL          ; auf naechste Elemente
            INC     DE          ; zeigen
            DJNZ    VERGL       ; gegebenenfalls gesamtes Feld
                                ; durchsuchen
FERTIG:     NOP                 ; gemeinsame Fortsetzungsstelle
```

3. Man muß hier die Aufgabenstellung anders interpretieren: Suche das erste Nibble mit dem Wert 0FH von hinten. Diese Aufgabe läßt sich sehr leicht aus dem Text herleiten (siehe die Routine für das Suchen einer 0 und die Vorlauf- beziehungsweise Nachlaufroutinen dazu). Man muß nur die Bearbeitungsrichtung ändern.

4. Die eigentliche Schwierigkeit lag hier bei den Feldgrenzen. Dabei kann man wie im Textbeispiel vorgehen. Der Schleifenkörper selbst gestaltet sich sehr einfach, indem man z.B. mit dem B-Register eine einfache Zählschleife aufbaut und im Körper lediglich ein Byte lädt, komplementiert und wieder abspeichert. Ein Programmlisting ist deshalb an dieser Stelle überflüssig.

Kapitel 14.4

1. Für die Lösung soll das HL-Register auf das letzte Feldelement zeigen und das BC-Register die Anzahl (> 1) der Feldelemente enthalten:

```
            LD      D,H         ; Register
            LD      E,L         ; kopieren
            DEC     DE          ; DE-Register wird Zielregister
            DEC     BC          ; das erste Feldelement
                                ; ist schon fertig
            LDDR                ; Feld initialisieren
```

2. HL- und DE-Register sollen jeweils auf das Ende der Bytefelder zeigen, das BC-Register enthält die Länge des Felds:

```
            LD      A,'0'       ; Vergleichswert laden
```

```
KOPIE:   CP      (HL)        ; mit aktueller Speicherzelle
                             ; vergleichen
         JP      Z,FERTIG    ; Zeichen gefunden
         LDD                 ; kopieren
         JP      PE,KOPIE    ; gegebenenfalls gesamtes Feld
                             ; bearbeiten
FERTIG:  NOP
```

3. Da das Schema der Routinen bereits vorgegeben ist und man dort nur die Bearbeitungsrichtung wechseln muß, wird an dieser Stelle auf eine detaillierte Lösung verzichtet.

4. Die wesentlichen Ideen kann man sich aus der Lösung des entsprechenden Problems für Nibblefelder holen; auf eine Ausarbeitung verzichte ich hier.

Kapitel 15.1

1. a) mit fester Länge 9:

```
TEXT1:   DEFM    'Eingabe'   ; Speicherbedarf 9 Bytes
TEXT2:   DEFM    'Guten Tag' ; Speicherbedarf 9 Bytes
                             ; (beachte, dass fuer das
                             ; Rufzeichen kein Platz mehr ist!)
TEXT3:   DEFM    'Seite 4'   ; Speicherbedarf 9 Bytes
                             ; (beachte, dass die
         DEFB    13          ; Steuerzeichen als Bytes
         DEFB    10          ; definiert wurden, aber trotzdem
                             ; zur Zeichenkette zaehlen)
```

b) mit Längenangabe:

```
TEXT1:   DEFB    7
         DEFM    'Eingabe'   ; Speicherbedarf 8 Bytes
```

oder

```
TEXT1:   DEFB    7
         DEFW    ADR1
         :
         :
ADR1:    DEFM    'Eingabe'   ; Speicherbedarf 10 Bytes
TEXT2:   DEFB    10
         DEFM    'Guten Tag!' ; Speicherbedarf 11 Bytes
```

```
TEXT3:  DEFB    9
        DEFM    'Seite 4'
        DEFB    13
        DEFB    10              ; Speicherbedarf 10 Bytes
```

c) mit Endemarkierung (00H):

```
TEXT1:  DEFM    'Eingabe'
        DEFB    00H             ; Speicherbedarf 8 Bytes
TEXT2:  DEFM    'Guten Tag!'
        DEFB    00H             ; Speicherbedarf 11 Bytes
TEXT3:  DEFM    'Seite 4'
        DEFB    13
        DEFB    10
        DEFB    00H             ; Speicherbedarf 10 Bytes
```

d) mit Endemarkierung und Längenbegrenzung 10:

```
TEXT1:  DEFM    'Eingabe'
        DEFB    00H             ; Speicherbedarf 10 Bytes
TEXT2:  DEFM    'Guten Tag!'    ; Speicherbedarf 10 Bytes
                                ; (beachte, dass keine
                                ; Endemarkierung erforderlich ist)
TEXT3:  DEFM    'Seite 4'
        DEFB    13
        DEFB    10
        DEFB    00H             ; Speicherbedarf 10 Bytes
```

Kapitel 15.2

1. Die Routine ist recht einfach (es entfällt sogar die vorher notwendige Erweiterung des Zählers):

```
KOPIE:  LD      C,(HL)          ; niederwertiges Byte
                                ; der Textlaenge holen
        INC     HL              ; auf hoeherwertiges Byte zeigen
        LD      B,(HL)          ; hoeherwertiges Byte
                                ; der Textlaenge holen
        DEC     HL              ; wieder auf Anfang
                                ; der Zeichenkette zeigen
        INC     BC              ; Laengenangabe mitrechnen
        INC     BC
        LDIR                    ; Transport durchfuehren
```

2. Man kann die im Text gezeigte Routine für dieses Problem mit leichten Modifikationen übernehmen. Man muß nur die automatische (äußere) Schleife in eine selbstgesteuerte Schleife mit dem BC-Register als Zähler umwandeln.

3. Hier gilt das gleiche wie bei der vorherigen Aufgabe: Die Schleife wird wieder anstatt mit dem B-Register mit dem BC-Register programmiert. Der Längenzähler wird nicht mehr im C-Register, sondern im DE-Register mitgeführt.

Kapitel 15.3

1. Für die Relationen < >bzw. >= ist nichts zu tun, da diese Fälle ja genau dem Gegenteil von = bzw. < entsprechen, es muß eben nur entsprechend anders verzweigt werden (zum Beispiel JP NZ anstelle von JP Z).
   Die Relation > erhält man aus der Relation < durch Vertauschen der Operanden, ebenso entsteht aus der Relation >= die Relation <=.

2. Bei diesem Programm gibt es drei Endekriterien, die bearbeitet werden müssen. Dabei werden zwei schon durch den CPI-Befehl ausgewertet, wenn man die Längenbegrenzung ins BC-Register bringt und die Endemarkierung (oder den Vergleichswert) vorher ins A-Register. Das HL-Register soll dabei wieder auf die Zeichenkette zeigen. Das Suchzeichen sei im A-Register.

```
ENDE     EQU     '$'          ; Endemarkierung
MAXL     EQU     512          ; maximale Laenge
         LD      BC,MAXL      ; Laenge ins Zaehlregister holen
         LD      D,A          ; Kopie des Suchzeichens
SUCHE:   LD      A,D          ; Suchzeichen holen
         CP      (HL)         ; Suchzeichen mit Zeichen
                              ; in Kette vergleichen
         JP      Z,GEFUND     ; Zeichen gefunden
         LD      A,ENDE       ; Endezeichen holen
         CPI                  ; Vergleich mit Endezeichen
                              ; und auf Laenge
         JP      PO,NGEFUN    ; Zeichenkette zu Ende
         JP      NZ,SUCHE     ; Zeichen nicht das Endezeichen,
NGEFUN:  INC     C            ; Null-Flag zuruecksetzen
GEFUND:  NOP                  ; gemeinsame Fortsetzungsstelle
```

Man erkennt am Null-Flag, ob die Suche erfolgreich war.

3. Bei dieser Aufgabe besteht das Hauptproblem darin, daß man – im Gegensatz zu den Zeichenketten mit Längenangabe – die Länge nicht im voraus weiß. Eine einfache Möglichkeit, das Problem anzugehen, wäre die folgende:

1. Längen der beiden Ketten bestimmen
2. Vergleichen, bei gleicher Länge 3. ausführen, sonst Problem gelöst
3. Zeichenketten vergleichen

Dabei läßt sich der erste Punkt mit Hilfe der im letzten Kapitel gezeigten Routinen bewältigen. Die Punkte 2 und 3 lassen sich dann genauso abwickeln, wie die im Text gezeigte Routine für die Kleiner-Relation der zweiten Art für Zeichenketten mit Längenangabe.

Kapitel 15.4

Die Register seien wieder so belegt wie im Text, das heißt, im HL-Register steht der Zeiger auf die Originalzeichenkette, im DE-Register der auf die Teilzeichenkette.

1. Der Markierungszeiger sei im BC-Register :

a) Zeichenkette mit Längenangabe:

```
ZEIGER:  DEFS    2           ; Hilfsspeicher
         LD      (ZEIGER),DE ; Zeiger sichern
         LD      E,(HL)      ; Laenge der
         INC     HL          ; Originalzeichenkette
         LD      D,(HL)      ; ins DE-Register laden
         INC     HL          ; HL zeigt nun auf das
                             ; erste Zeichen der Kette
         ADD     HL,DE       ; hinter die Zeichenkette zeigen
         OR      A           ; Uebertrag-Flag loeschen
         SBC     HL,BC       ; Laenge der Kopie errechnen
         LD      DE,(ZEIGER) ; Zeiger auf Kopie holen
         LD      A,L         ; niederwertiges Byte der Laenge
         LD      (DE),A      ; Laengenangabe fuer Kopie
                             ; erstellen
         LD      L,C         ; Tausch von C mit L
         LD      C,A
         INC     DE          ; fuer hoeherwertiges Byte ebenso
         LD      A,H
         LD      (DE),A
         LD      H,B
         LD      B,A
         INC     DE          ; jetzt ist die folgende
                             ; Zeigerverteilung erreicht:
                             ; <HL> => Originalteilzeichenkette
                             ; <DE> => Kopie
```

```
                                    ; <BC> = Länge der Teilzeichenkette
        DIR                         ; Teilzeichenkette kopieren
```

b) Zeichenkette mit Endemarkierung

```
ENDE    EQU     '$'         ; Endemarkierung
KOPIER: LD      A,(BC)      ; Zeichen aus Originalkette laden
        LD      (DE),A      ; in Kopie hinterlegen
        CP      ENDE        ; mit Endezeichen vergleichen
                NZ,KOPIER   ; weiterkopieren
```

2. Das BC-Register enthalte nun die Position n:

```
ZEIGER: DEFS    2           ; Hilfsspeicher
        LD      (ZEIGER),DE ; Zeiger auf Kopie sichern
        LD      E,(HL)      ; urspruengliche Laenge laden
        INC     HL
        LD      D,(HL)
        ADD     HL,BC       ; auf erstes zu kopierendes
                            ; Byte zeigen
        EX      DE,HL       ; Zeiger tauschen
        OR      A           ; Uebertrag-Flag loeschen
        SBC     HL,BC       ; Laenge der
        INC     HL          ; Kopie berechnen
        LD      B,H         ; Laenge ins BC-Register
        LD      C,L         ; als Zaehler bringen
        LD      HL,(ZEIGER) ; Zeiger auf Kopie holen
        LD      (HL),C      ; Laenge dort hinterlegen
        INC     HL
        LD      (HL),B
        INC     HL          ; auf das erste Byte zeigen
        EX      DE,HL       ; Zeiger vertauschen
        LDIR                ; kopieren
```

3. In diesem Fall erleichtert die Endemarkierung das Kopieren nicht, es muß die Länge der Teilzeichenkette errechnet werden:

```
ENDE    EQU     '$'         ; Endezeichen
ZEIGER: DEFS    2           ; Hilfsspeicher
M:      DEFS    2           ; hier muss m stehen
N:      DEFS    2           ; hier muss n stehen
        LD      BC,(M)      ; Index des ersten Zeichens laden
        ADD     HL,BC       ; Zeiger errechnen
```

```
        DEC     HL
        LD      (ZEIGER),HL     ; Adresse sichern
        LD      HL,(N)          ; Index des letzten Zeichens laden
        OR      A               ; Uebertrag-Flag loeschen
        SBC     HL,BC           ; Differenz m-n+1
        INC     HL              ; ausrechnen (= Laenge der Kopie)
        LD      B,H             ; Laenge in Zaehler
        LD      C,L             ; bringen
        LD      HL,(ZEIGER)     ; Zeiger auf Original holen
        LDIR                    ; kopieren
        LD      A,ENDE          ; Endezeichen laden
        LD      (DE),A          ; Endemarkierung setzen
```

Kapitel 15.5

1. Die einfachste Lösungsidee besteht darin, die Längen der Zeichenketten zu bestimmen und dann wie bei dem Beispielprogramm im Text fortzufahren (es entfällt dann sogar die Berechnung der neuen Länge). Die Längenbestimmung ist unkompliziert und läßt sich mit den bereits gezeigten Methoden durchführen.

2. Eine mögliche Lösung (die Zeiger seien wie bei dem Textbeispiel):

```
ZEIGER:  DEFS    2               ; Hilfsspeicher
LDIFF:   DEFS    2               ; Hilfsspeicher
TKETTE:  DEFS    2               ; Hilfsspeicher
         LD      (ZEIGER),HL     ; Zeiger zwischenspeichern
         D       H,B             ; Zeiger hinter zu loeschende
         LD      L,C             ; Teilzeichenkette kopieren
         OR      A               ; Uebertrag-Flag loeschen
         SBC     HL,DE           ; Laenge der zu loeschenden
                                 ; Kette ausrechnen
         LD      (LDIFF),HL      ; und sichern
         LD      (TKETTE),DE     ; Zeiger auf Teilzeichenkette
                                 ; sichern
         LD      E,(HL)          ; Laenge
         INC     HL              ; der
         LD      D,(HL)          ; Zeichenkette holen
         INC     HL              ; auf erstes Zeichen zeigen
         ADD     HL,DE           ; Laenge des Rests
         OR      A               ; berechnen, der auf
         SBC     HL,BC           ; die Teilzeichenkette
         EX      DE,HL           ; folgt
```

```
LD      H,B             ; Vorbereitungen
LD      L,C             ; für
LD      B,D             ; den
LD      C,E             ; Blockkopierbefehl
LD      DE,(TKETTE)     ; Zeiger auf Teilzeichenkette
                        ; restaurieren
LDIR                    ; kopieren und loeschen
LD      HL,(ZEIGER)     ; Zeiger auf Kette holen
LD      BC,(LDIFF)      ; Laengendifferenz holen
LD      E,(HL)          ; urspruengliche Laenge holen
INC     HL
LD      D,(HL)
EX      DE,HL           ; Zeiger sichern
OR      A               ; Uebertrag-Flag loeschen
SBC     HL,BC           ; neue Laenge der Kette berechnen
EX      DE,HL           ; Zeiger wieder holen
LD      (HL),D          ; neue Laenge eintragen
DEC     HL
LD      (HL),E
```

3. Diese Aufgabe kann prinzipiell auf zwei verschiedene Arten gelöst werden. Die einfachste Methode besteht darin, mit den bereits vorhandenen Programmen die zu ersetzende Teilzeichenkette zu löschen und danach die andere Teilzeichenkette einzufügen. Diese Vorgehensweise hat allerdings (insbesonders bei großen Zeichenketten) den Nachteil, daß zweimal eine Blockverschiebung durchgeführt werden muß. Eine elegantere Lösung ist es, zunächst die Differenz der Längen der Teilzeichenketten zu berechnen. Dann wird die Blockverschiebung der restlichen Zeichenkette so durchgeführt, daß die einzufügende Zeichenkette genau Platz hat. Diese kopiert man dann in diesen Bereich hinein.

   Für die erste Methode erübrigt sich ein Programm, da die Teile schon vorhanden sind. Für die zweite Methode kann man in etwa die normale Einfügeroutine verwenden, lediglich der Anfang muß so verändert werden, daß man in beide Richtungen verschieben kann. Deshalb ist ein Programm hier überflüssig.

Kapitel 16.1

1. Wir nehmen an, daß die Zeichenketten Längenangaben besitzen. Bei der Operation »Gleich« vergleichen wir zunächst die Längenangaben. Differieren diese, so sind die Mengen mit Sicherheit verschieden. Andernfalls prüfen wir nach, ob jedes Element der einen Menge in der anderen Menge enthalten ist.

   HL und DE zeigen auf die beiden Mengen. Wenn die Mengen übereinstimmen, soll das Null-Flag gesetzt werden. Den Zeiger auf die Längenangabe der zweiten Menge müssen wir in einer Hilfsvariablen unterbringen.

```
                                        ; Programmbereich
        LD      A,(DE)          ; Kardinalitaet der ersten Menge
                                ; holen
        CP      (HL)            ; mit Kardinalitaet der zweiten
                                ; Menge vergleichen
        JP      NZ,FERTIG       ; Mengen sind verschieden
        LD      C,A             ; Zaehler fuer erste Menge
                                ; aufsetzen
        INC     C               ; Korrektur fuer abweisende
                                ; Schleife
        LD      (MENGE2),HL     ; Zeiger auf zweite Menge sichern
        JP      GEFUND          ; in abweisende Schleife
                                ; einspringen
HOLE:   INC     DE              ; auf naechstes Element der
                                ; ersten Menge zeigen
        LD      A,(DE)          ; Element der ersten Menge holen
        LD      HL,(MENGE2)     ; Zeiger auf zweite Menge holen
        LD      B,(HL)          ; Zaehler fuer zweite Menge
                                ; aufsetzen
PRUEFE: INC     HL              ; auf naechstes Element der
                                ; zweiten Menge zeigen
        CP      (HL)            ; mit Element der ersten Menge
                                ; vergleichen
        JP      Z,GEFUND        ; Elemente stimmen ueberein
        DJNZ    PRUEFE          ; gegebenenfalls alle Elemente
                                ; der zweiten Menge ansehen
        JP      FERTIG          ; Mengen sind nicht gleich
GEFUND: DEC     C               ; Anzahl der restlichen Elemente
                                ; der ersten Menge berechnen
        JP      NZ,HOLE         ; alle Elemente der ersten Menge
                                ; bearbeiten
FERTIG: NOP                     ; gemeinsame Fortsetzungsstelle
                                ; Datenbereich
MENGE2: DEFS    2               ; Zeiger auf Laengenangabe der
                                ; zweiten Menge
```

Bei der Anzahl der Operationen interessieren wir uns nur für die eigentlichen Vergleichsoperationen zwischen Elementen der beiden Mengen. Differieren die Kardinalitäten, so werden keine Elemente verglichen; die minimale Anzahl von Operationen ist also Null. Stimmen dagegen die Kardinalitäten der beiden Mengen überein (die Kardinalität sei dann n), und haben sie genau n-1 gemeinsame Elemente, wobei das letzte Element der ersten Menge nicht in der zweiten ist, und das erste Element der zweiten Menge nicht in der ersten, so braucht man die maximale Anzahl von $(n^2+3^{n-2})/2$ Operationen.

Bei der Operation »Teilmenge von« prüfen wir für jedes Element der ersten Menge, ob es in der zweiten Menge enthalten ist. Zuvor stellen wir fest, ob die erste Menge mehr Elemente als die zweite enthält, in welchem Fall die Relation »Teilmenge von« nicht gegeben sein kann.

Das DE-Register zeigt auf die erste Menge, das HL-Register auf die zweite. Wenn die erste Menge Teilmenge der zweiten Menge ist, so wird das Null-Flag gesetzt. Der Algorithmus unterscheidet sich nur durch das anfängliche Prüfen der Kardinalitäten vom vorhergehenden.

```
                                    ; Programmbereich
        EX      DE,HL               ; Zeiger tauschen
        LD      A,(DE)              ; Kardinalitaet der zweiten Menge
                                    ; holen
        CP      (HL)                ; mit Kardinalitaet der ersten
                                    ; Menge vergleichen
        JP      C,FERTIG            ; erste Menge nicht Teilmenge
                                    ; der zweiten Menge
        LD      C,(HL)              ; Zaehler fuer erste Menge
                                    ; aufsetzen
        INC     C                   ; Korrektur fuer abweisende
                                    ; Schleife
        EX      DE,HL               ; Zeiger tauschen
        LD      (MENGE2),HL         ; Zeiger auf zweite Menge sichern
        JP      GEFUND              ; in abweisende Schleife
                                    ; einspringen
HOLE:   INC     DE                  ; auf naechstes Element der
                                    ; ersten Menge zeigen
        LD      A,(DE)              ; Element der ersten Menge holen
        LD      HL,(MENGE2)         ; Zeiger auf zweite Menge holen
        LD      B,(HL)              ; Zaehler fuer zweite Menge
                                    ; aufsetzen
PRUEFE: INC     HL                  ; auf naechstes Element der
                                    ; zweiten Menge zeigen
        CP      (HL)                ; mit Element der ersten Menge
                                    ; vergleichen
        JP      Z,GEFUND            ; Elemente stimmen ueberein
        DJNZ    PRUEFE              ; gegebenenfalls alle Elemente
                                    ; der zweiten Menge ansehen
        JP      FERTIG              ; erste Menge nicht Teilmenge
                                    ; der zweiten Menge
GEFUND: DEC     C                   ; Anzahl der restlichen Elemente
                                    ; der ersten Menge berechnen
        JP      NZ,HOLE             ; alle Elemente der ersten Menge
                                    ; bearbeiten
FERTIG: NOP                         ; gemeinsame Fortsetzungsstelle
```

```
                                    ; Datenbereich
MENGE2:  DEFS       2               ; Zeiger auf Laengenangabe der
                                    ; zweiten Menge
```

Übersteigt die Kardinalität der ersten Menge die Kardinalität der zweiten Menge, so ist wieder kein Vergleich notwendig. Wir bezeichnen mit m die Kardinalität der ersten Menge, mit n die Kardinalität der zweiten Menge. Stimmen die ersten m-1 Elemente der ersten Menge mit den letzten m-1 Elementen der zweiten Menge überein und ist das letzte Element der ersten Menge nicht in der zweiten Menge enthalten, so wird die maximale Anzahl von m_n-(m-2) (m-1)/2 Vergleichen durchgeführt.

Bei der Operation »Element von« nehmen wir an, daß im A-Register das gesuchte Element steht und daß das HL-Register auf die Menge zeigt. Ist das Element in der Menge enthalten, so wird das Null-Flag gesetzt.

```
         LD         C,(HL)      ; Kardinalitaet der Menge holen
         DEC        C           ; auf leere Menge testen
         JP         M,FERTIG    ; leere Menge,
                                ; Relation nicht erfuellt
         INC        C           ; Kardinalitaet der Menge
                                ; in Zaehler laden
         LD         B,0
         INC        HL          ; auf erstes Element zeigen
         CPIR                   ; nach Element suchen
FERTIG:  NOP                    ; gemeinsame Fortsetzungsstelle
```

Bei leerer Menge führen wir keinen Vergleich durch. Ist das Element nicht in der Menge enthalten, so stimmt die Anzahl der Operationen mit der Kardinalität n überein.

2. Die Mengen sollen aufsteigend geordnet sein. Alle Rahmenbedingungen sind wie in der ersten Aufgabe.

   Beim Test auf Gleichheit prüfen wir wieder zuerst die Übereinstimmung der Kardinalitäten. Anschließend gehen wir beide Mengen simultan elementweise durch und testen auf paarweise Übereinstimmung der einzelnen Elemente.

```
         LD         A,(DE)      ; Kardinalitaet der ersten Menge
                                ; holen
         LD         B,A         ; Kardinalitaet der ersten Menge
                                ; in Zaehlgroesse bringen
         INC        B           ; und fuer abweisende Schleife
                                ; korrigieren
         CP         (HL)        ; mit Kardinalitaet der zweiten
                                ; Menge vergleichen
         JP         NZ,FERTIG   ; Mengen sind verschieden
```

```
                    TEST        ; in abweisende Schleife springen
PRUEFE:   INC       DE          ; auf naechstes Element der
                                ; ersten Menge zeigen
          INC       HL          ; auf naechstes Element der
                                ; zweiten Menge zeigen
          D         A,(DE)      ; Element der ersten Menge holen
          CP        (HL)        ; mit Element der zweiten Menge
                                ; vergleichen
          JP        NZ,FERTIG   ; Mengen sind verschieden
TEST:     DJNZ      PRUEFE      ; gegebenenfalls alle Elemente
                                ; vergleichen
FERTIG:   NOP                   ; gemeinsame Fortsetzungsstelle
```

Die minimale Anzahl von Operationen ist offensichtlich wieder Null, die maximale stimmt mit der Kardinalität n der beiden Mengen überein, falls beide Mengen gleichmächtig sind. Die Anzahl der Operationen ist hier also wesentlich kleiner als in der Lösung mit ungeordneten Mengen (linear im Gegensatz zu quadratisch in der Kardinalität); dies wird auch durch die Tatsache, daß eine Vergleichsoperation hier fast doppelt so aufwendig ist wie in der ersten Aufgabe, nicht wesentlich beeinflußt.

Beim Test auf Teilmenge vergleichen wir wieder zunächst die Kardinalitäten. Dann prüfen wir, ob jedes Element der ersten Menge in der zweiten Menge enthalten ist. Den Zeiger auf das jeweils nächste Element der zweiten Menge können wir wegen der Ordnung der Mengen stets beibehalten; jedes Element der ersten Menge wird damit höchstens einmal angeschaut.

```
          EX        DE,HL       ; Zeiger tauschen
          LD        A,(DE)      ; Kardinalitaet der zweiten Menge
                                ; holen
          CP        (HL)        ; mit Kardinalitaet der ersten
                                ; Menge vergleichen
          JP        C,FERTIG    ; erste Menge nicht Teilmenge
                                ; der zweiten Menge
          LD        B,A         ; Zaehler fuer zweite Menge
                                ; aufsetzen
          LD        C,(HL)      ; Zaehler fuer erste Menge
                                ; aufsetzen
          INC       C           ; Korrektur fuer abweisende
                                ; Schleife
          EX        DE,HL       ; Zeiger tauschen
          JP        GEFUND      ; in abweisende Schleife
                                ; einspringen
HOLE:     INC       DE          ; auf naechstes Element der
                                ; ersten Menge zeigen
          LD        A,(DE)      ; Element der ersten Menge holen
```

```
            INC     B           ; Schleife abweisend
            JP      TEST        ; machen
PRUEFE:     INC     HL          ; auf naechstes Element der
                                ; zweiten Menge zeigen
            CP      (HL)        ; mit Element der ersten Menge
                                ; vergleichen
            JP      Z,GEFUND    ; Elemente stimmen ueberein
TEST:       DJNZ    PRUEFE      ; gegebenenfalls alle Elemente
                                ; der zweiten Menge ansehen
            JP      FERTIG      ; erste Menge nicht Teilmenge
                                ; der zweiten Menge
GEFUND:     DEC     B           ; Zaehler fuer zweite Menge
                                ; korrigieren
            DEC     C           ; Anzahl der restlichen Elemente
                                ; der ersten Menge berechnen
            P       NZ,HOLE     ; alle Elemente der ersten Menge
                                ; bearbeiten
FERTIG:     NOP                 ; gemeinsame Fortsetzungsstelle
```

Stimmen die Kardinalitäten nicht überein, so wird keine Operation durchgeführt. Die Maximalzahl von Vergleichen ist gleich der Kardinalität n der zweiten Menge. Dies ist – trotz wesentlich aufwendigerem Vergleichsmechanismus – wieder eine Größenordnung weniger als in der Lösung der ersten Aufgabe.

Durch eine kleine Modifikation des Vergleichs können wir in der Regel einen schnelleren Abbruch des Verfahrens erwarten, wenn die erste Menge nicht Teilmenge der zweiten Menge ist. Wenn nämlich das gerade getestete Element der zweiten Menge bereits größer als das aktuelle Element der ersten Menge ist, so garantiert die aufsteigende Ordnung (der obenstehende Algorithmus funktioniert ohne Modifikation auch für absteigende Ordnung), daß das Element der ersten Menge nicht in der zweiten enthalten ist, da in der zweiten Menge ab jetzt nur noch größere Elemente folgen können. Die Modifikation vergrößert allerdings den Aufwand für einen Vergleich ein bißchen.

```
            EX      DE,HL       ; Zeiger tauschen
            LD      A,(DE)      ; Kardinalitaet der zweiten Menge
                                ; holen
            CP      (HL)        ; mit Kardinalitaet der ersten
                                ; Menge vergleichen
            JP      C,FERTIG    ; erste Menge nicht Teilmenge
                                ; der zweiten Menge
            LD      B,A         ; Zaehler fuer zweite Menge
                                ; aufsetzen
            LD      C,(HL)      ; Zaehler fuer erste Menge
                                ; aufsetzen
```

```
            INC     C           ; Korrektur fuer abweisende
                                ; Schleife
            EX      DE,HL       ; Zeiger tauschen
            JP      GEFUND      ; in abweisende Schleife
                                ; einspringen
HOLE:       INC     DE          ; auf naechstes Element der
                                ; ersten Menge zeigen
            LD      A,(DE)      ; Element der ersten Menge holen
            INC     B           ; Schleife abweisend
            JP      TEST        ; machen
PRUEFE:     INC     HL          ; auf naechstes Element der
                                ; zweiten Menge zeigen
            CP      (HL)        ; mit Element der ersten Menge
                                ; vergleichen
            JP      Z,GEFUND    ; Elemente stimmen ueberein
            JP      C,FERTIG    ; erste Menge nicht Teilmenge
                                ; der zweiten Menge
TEST:       DJNZ    PRUEFE      ; gegebenenfalls alle Elemente
                                ; der zweiten Menge ansehen
            JP      FERTIG      ; erste Menge nicht Teilmenge
                                ; der zweiten Menge
GEFUND:     DEC     B           ; Zaehler fuer zweite Menge
                                ; korrigieren
            DEC     C           ; Anzahl der restlichen Elemente
                                ; der ersten Menge berechnen
            JP      NZ,HOLE     ; alle Elemente der ersten Menge
                                ; bearbeiten
FERTIG:     NOP                 ; gemeinsame Fortsetzungsstelle
```

Zum Überprüfen der Relation »Element von« können wir uns der gleichen Routine bedienen wie in Aufgabe 1. Der Aufwand wird dabei aber durch die vorgegebene Ordnung nicht reduziert. Der folgende Algorithmus macht von der Ordnung Gebrauch, reduziert aber nicht die maximale, sondern die durchschnittliche Anzahl von Vergleichen.

```
            LD      B,(HL)      ; Kardinalitaet der Menge in
                                ; Zaehlgroesse laden
            INC     B           ; Schleife abweisend machen
            JP      TEST
PRUEFE:     INC     HL          ; auf naechstes Element der
                                ; Menge zeigen
            CP      (HL)        ; mit vorgegebenem Element
                                ; vergleichen
            JP      Z,FERTIG    ; Element gefunden
```

```
        JP      C,FERTIG    ; Element nicht in der Menge
TEST:   DJNZ    PRUEFE      ; gegebenenfalls alle Elemente
                            ; der Menge vergleichen
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Kapitel 16.2

1. Die Menge der kleinen Buchstaben hat 26 Elemente. Wir benötigen für jeden Inzidenzvektor also 26 Bits. Zur Vereinfachung der Operationen vergeben wir jeweils 4 Bytes und verschenken die nicht benötigten 6 Bits.

   Folgende Routine bildet die Vereinigung zweier Mengen. DE und HL sind Zeiger auf die Operanden, BC ist Zeiger auf das Ergebnis.

```
        LD      A,(DE)      ; 8 Elemente holen
        OR      (HL)        ; Vereinigung bilden
        LD      (BC),A      ; 8 Elemente abspeichern
        INC     HL          ; auf die naechsten 8 Elemente
                            ; zeigen
        INC     DE
        INC     BC
        LD      A,(DE)      ; 8 Elemente holen
        OR      (HL)        ; Vereinigung bilden
        LD      (BC),A      ; 8 Elemente abspeichern
        INC     HL          ; auf die naechsten 8 Elemente
                            ; zeigen
        INC     DE
        INC     BC
        LD      A,(DE)      ; 8 Elemente holen
        OR      (HL)        ; Vereinigung bilden
        LD      (BC),A      ; 8 Elemente abspeichern
        INC     HL          ; auf die naechsten 2 Elemente
                            ; zeigen
        INC     DE
        INC     BC
        LD      A,(DE)      ; 2 Elemente holen
        OR      (HL)        ; Vereinigung bilden
        LD      (BC),A      ; 2 Elemente abspeichern
```

Zur Bildung des Schnitts ersetzen wir nur die OR-Befehle durch AND-Befehle. Bei der Bildung der Differenz zeigt HL auf den ersten Operanden, DE auf den zweiten Operanden, BC auf das Ergebnis.

```
        LD      A,(DE)      ; 8 Elemente holen
        CPL                 ; Komplement bilden
        AND     (HL)        ; Schnittmenge bilden
        LD      (BC),A      ; 8 Elemente abspeichern
        INC     HL          ; auf die naechsten 8 Elemente
                            ; zeigen
        INC     DE
        INC     BC
        LD      A,(DE)      ; 8 Elemente holen
        CPL                 ; Komplement bilden
        AND     (HL)        ; Schnittmenge bilden
        LD      (BC),A      ; 8 Elemente abspeichern
        INC     HL          ; auf die naechsten 8 Elemente
                            ; zeigen
        INC     DE
        INC     BC
        LD      A,(DE)      ; 8 Elemente holen
        CPL                 ; Komplement bilden
        AND     (HL)        ; Schnittmenge bilden
        LD      (BC),A      ; 8 Elemente abspeichern
        INC     HL          ; auf die naechsten 2 Elemente
                            ; zeigen
        INC     DE
        INC     BC
        LD      A,(DE)      ; 2 Elemente holen
        CPL                 ; Komplement bilden
        AND     (HL)        ; Schnittmenge bilden
        LD      (BC),A      ; 2 Elemente abspeichern
```

2. Die Menge der 7-Bit-ASCII-Zeichen enthält 128 Elemente. Jeder Inzidenzvektor ist damit 128 Bits = 16 Bytes lang. Wir bauen eine Zählschleife mit 16 Durchläufen auf.

   Zunächst bilden wir das Komplement der Menge, auf die das HL-Register zeigt. DE verweist auf das Ergebnis.

```
        LD      B,16        ; Anzahl der Schleifendurchlaeufe
KOMPL:  LD      A,(HL)      ; 8 Elemente holen
        CPL                 ; Komplement bilden
        LD      (DE),A      ; 8 Elemente abspeichern
        INC     HL          ; auf die naechsten 8 Elemente
                            ; zeigen
        INC     DE
        DJNZ    KOMPL       ; gesamten Inzidenzvektor
                            ; bearbeiten
```

Bei der Bildung der symmetrischen Differenz zeigen HL und DE auf die beiden Operanden, BC auf das Ergebnis. Wir können deswegen das B-Register nicht mehr als Zählgröße verwenden; statt dessen legen wir eine Zählvariable ZAEHL an.

```
                                    ; Programmbereich
        LD      A,16        ; Anzahl der Schleifendurchlaeufe
        LD      (ZAEHL),A   ; Zaehlgroesse aufsetzen
SYMDIF: LD      A,(DE)      ; 8 Elemente holen
        XOR     (HL)        ; symmetrische Differenz bilden
        LD      (BC),A      ; 8 Elemente abspeichern
        INC     HL          ; auf die naechsten 8 Elemente
                            ; zeigen
        INC     DE
        INC     BC
                A,(ZAEHL)   ; Zaehlgroesse holen
        DEC     A           ; Anzahl der restlichen
                            ; Durchlaeufe berechnen
        LD      (ZAEHL),A   ; Zaehlgroesse abspeichern
        JP      NZ,SYMDIF   ; Inzidenzvektoren vollstaendig
                            ; bearbeiten
                            ; Datenbereich
ZAEHL:  DEFS    1           ; Zaehlgroesse
```

3. Bei additiver Mischung gibt es 7 Möglichkeiten, die Farben Violett, Grün und Orange zu mischen:

violett	grün	orange	Resultat
ja	nein	nein	Violett
nein	ja	nein	Grün
ja	ja	nein	Blau
nein	nein	ja	Orange
ja	nein	ja	Rot
nein	ja	ja	Gelb
ja	ja	ja	Weiß

Bei subtraktiver Mischung gibt es 7 Möglichkeiten, die Farben Rot, Gelb und Blau zu mischen:

rot	gelb	blau	Resultat
ja	nein	nein	Rot
nein	ja	nein	Gelb

ja	ja	nein	Orange
nein	nein	ja	Blau
ja	nein	ja	Violett
nein	ja	ja	Grün
ja	ja	ja	Schwarz

Wir ordnen jeder Menge von Farben einen Inzidenzvektor zu, in welchem die Farben folgendermaßen codiert sind:

Bit	Farbe
0	Violett
1	Grün
2	Orange
3	Rot
4	Gelb
5	Blau
6	Weiß
7	Schwarz

Zunächst zur additiven Mischung. Als Eingabe erwarten wir einen Inzidenzvektor, in dem eine oder mehrere Farben aus der Palette Violett, Grün, Orange dargestellt sind. Dies bedeutet bei der gewählten Codierung, daß nur die Werte 1 bis 7 für den Inzidenzvektor zulässig sind. Wir legen eine Liste an, aus der wir die jeweilige Resultatfarbe (wieder als Inzidenzvektor) heraussuchen können, wobei der eingegebene Inzidenzvektor als Index dient. Operand und Ergebnis soll das A-Register sein.

```
                                ; Programmbereich
OR      A                       ; auf leere Menge testen
JP      Z,FEHLER                ; unzulaessiger Inzidenzvektor
CP      8                       ; auf unzulaessige Farben testen
JP      NC,FEHLER               ; unzulaessiger Inzidenzvektor
LD      HL,AFARBE-1             ; auf fiktives nulltes Element
                                ; der Liste der Resultatfarben
                                ; zeigen
LD      D,0                     ; Index zu Relativadresse machen
LD      E,A
ADD     HL,DE                   ; Adresse der Resultatfarbe
                                ; berechnen
LD      A,(HL)                  ; Resultatfarbe holen
                                ; Datenbereich
```

```
AFARBE:  DEFB    00000001B    ; Violett
         DEFB    00000010B    ; Gruen
         DEFB    00100000B    ; Blau
         DEFB    00000100B    ; Orange
         DEFB    00001000B    ; Rot
         DEFB    00010000B    ; Gelb
         DEFB    01000000B    ; Weiss
```

Bei der subtraktiven Mischung gehen wir genauso vor; jedoch müssen wir zuvor den Inzidenzvektor der Mischung dreimal zirkulär rechtsrotieren, um die Farben Rot, Gelb, Blau auf die Indexpositionen 0, 1, 2 zu bringen.

```
                                  ; Programmbereich
         RRCA                     ; Farben Rot, Gelb, Blau auf
                                  ; Indexpositionen 0, 1, 2 bringen
         RRCA
         RRCA
         OR      A                ; auf leere Menge testen
         JP      Z,FEHLER         ; unzulaessiger Inzidenzvektor
         CP      8                ; auf unzulaessige Farben testen
         JP      NC,FEHLER        ; unzulaessiger Inzidenzvektor
         LD      HL,SFARBE-1      ; auf fiktives nulltes Element
                                  ; der Liste der Resultatfarben
                                  ; zeigen
         LD      D,0              ; Index zu Relativadresse machen
         LD      E,A
         ADD     HL,DE            ; Adresse der Resultatfarbe
                                  ; berechnen
         LD      A,(HL)           ; Resultatfarbe holen
                                  ; Datenbereich
SFARBE:  DEFB    00001000B        ; Rot
         DEFB    00010000B        ; Gelb
         DEFB    00000100B        ; Orange
         DEFB    00100000B        ; Blau
         DEFB    00000001B        ; Violett
         DEFB    00000010B        ; Gruen
         DEFB    10000000B        ; Schwarz
```

Kapitel 17.1

1. Für alle drei Größen reicht je ein Byte aus. Es ergibt sich folgende Datenstruktur (die Größe BASIS stellt die Basis-Adresse des Verbunds dar):

```
BASIS:
ALTER:    DEFS    1          ; Alter (in Jahren)
GROESS:   DEFS    1          ; Groesse (in cm)
GEWICH:   DEFS    1          ; Gewicht (in kg)
```

Das folgende Programmstück liefert die Abweichung vom Normalgewicht (tatsächliches Gewicht - Normalgewicht) im D-Register. Wenn alle genannten Bedingungen erfüllt sind, wird zu einer Adresse JA verzweigt, sonst zu einer Adresse NEIN.

```
          LD      IX,BASIS          ; Basis-Adresse
          LD      A,(IX+GEWICH-BASIS)    ; Gewicht holen
          ADD     A,100             ; Normalgewicht subtrahieren
          SUB     (IX+GROESS-BASIS)
          LD      D,A               ; Abweichung vom Normalgewicht
                                    ; sichern
          LD      A,55              ; vorgegebenes Alter
          CP      (IX+ALTER-BASIS)  ; mit tatsaechlichem
                                    ; Alter vergleichen
          JP      NC,NEIN           ; nicht aelter als 55 Jahre
          CP      182               ; mit vorgegebener Groesse
                                    ; vergleichen
          JP      NC,NEIN           ; nicht kleiner als 182 cm
          LD      A,(IX+GEWICH-BASIS)    ; Gewicht holen
                  78                ; mit vorgegebenem Gewicht
                                    ; vergleichen
          JP      C,NEIN            ; weniger als 78 Kilo schwer
          JP      JA                ; alle Bedingungen erfuellt
```

Das Programmstück könnte noch optimiert werden, wenn Größe und Gewicht in Hilfsregistern zwischengespeichert würden.

2. Die Position in Zeile und Spalte wird jeweils ab Null gezählt. Bei Erreichen des Zeilenendes wird auf Spalte Null gesprungen; außerdem wird ein Zeilenvorschub nötig. Bei Erreichen des Seitenendes wird auf Zeile Null gesprungen. Die Größe stellt die Basisadresse des Verbunds dar:

```
          LD      IX,BASIS          ; Basis-Adresse
          INC     (IX+ZPOSIT-BASIS) ; Zeilenposition
                                    ; fortschalten
          LD      A,(IX+ZPOSIT-BASIS) ; neue Spalte holen
          P       (IX+ZLAENG-BASIS) ; mit maximaler
                                    ; Zeilenlaenge
                                    ; vergleichen
```

```
        JP      NZ,FERTIG           ; kein Zeilenvorschub
                (IX+ZPOSIT-BASIS),0 ; Zeilenposition
                                    ; ruecksetzen
        INC     (IX+SPOSIT-BASIS)   ; Seitenposition
                                    ; fortschalten
        LD      A,(IX+SPOSIT-BASIS) ; neue Zeile holen
        CP      (IX+SLAENG-BASIS)   ; mit maximaler
                                    ; Seitenlaenge
                                    ; vergleichen
        JP      NZ,FERTIG           ; kein Seitenvorschub
        LD      (IX+SPOSIT-BASIS),0 ; Seitenposition
                                    ; ruecksetzen
FERTIG: NOP                         ; Fortsetzungsstelle
```

3. Wir nehmen an, daß das Feld einen Deskriptor besitzt, der die Anzahl der Feldelemente angibt. Den am weitesten entfernten Punkt finden wir, indem wir das Quadrat des Abstands berechnen, nach dem Satz von Pythagoras: X*X + Y*Y. Wir verwenden einen Multiplikationsalgorithmus aus Kapitel 13.1; Multiplikator und Multiplikand machen wir zuvor positiv.

   Zur Durchführung des Verfahrens müssen wir einige Informationen in Variablen abspeichern. Der Zeiger auf den bisher am weitesten entfernten Punkt steht in der Variablen PUNKT, das Quadrat des zugehörigen Abstands in der Variablen ABSTND. Das Quadrat von X speichern wir als Zwischenergebnis in der Variablen XQUADR.

   Die Routine erhält im IX-Register einen Zeiger auf das Feld von Punkten. Nach Abschluß des Verfahrens befindet sich die X-Koordinate des gesuchten Punkts im B-Register, die Y-Koordinate im C-Register und das Quadrat des Abstands im DE-Register.

```
                                    ; Datenbereich
PUNKT:  DEFS    2                   ; Zeiger auf bisher weitest
                                    ; entfernten Punkt
ABSTND: DEFS    2                   ; Quadrat des bisher groessten
                                    ; Abstands
XQUADR: DEFS    2                   ; Quadrat der X-Koordinate
                                    ; Programmbereich
WPUNKT: LD      C,(IX+0)            ; Anzahl der Punkte holen
        INC     C                   ; auf Anzahl Null testen
        DEC     C
        JP      Z,FEHLER            ; kein Punkt vorhanden,
                                    ; Aufgabe sinnlos gestellt
        LD      HL,0                ; Quadrat des Abstands
                                    ; initialisieren
        LD      (ABSTND),HL
```

```
PRUEFE:  LD      A,(IX+1)        ; X-Koordinate holen
         OR      A               ; Vorzeichen testen
         JP      P,POSIT1        ; positives Vorzeichen
         NEG                     ; Betrag bilden
POSIT1:  LD      HL,0            ; Akkumulator loeschen
         LD      D,0             ; Multiplikand = Multiplikator
         LD      E,A
         LD      B,8             ; Schleifenzaehler mit Laenge
                                 ; des Multiplikators (in Bits)
                                 ; besetzen
MULTI1:  ADD     HL,HL           ; Akkumulator verdoppeln
         RLCA                    ; hoechstes Bit des
                                 ; Multiplikators in
                                 ; Uebertrag-Flag bringen,
                                 ; gleichzeitig Multiplikator
                                 ; zirkulaer links rotieren
         JP      NC,NULL1        ; keine Addition erforderlich
         ADD     HL,DE           ; Multiplikand zu Akkumulator
                                 ; addieren
NULL1:   DJNZ    MULTI1          ; alle Bits des Multiplikators
                                 ; verarbeiten
         LD      (XQUADR),HL     ; Quadrat der X-Koordinate
                                 ; abspeichern
         LD      A,(IX+2)        ; Y-Koordinate holen
         OR      A               ; Vorzeichen testen
         JP      P,POSIT2        ; positives Vorzeichen
         NEG                     ; Betrag bilden
POSIT2:  LD      HL,0            ; Akkumulator loeschen
         LD      D,0             ; Multiplikand = Multiplikator
         LD      E,A
         LD      B,8             ; Schleifenzaehler mit Laenge
                                 ; des Multiplikators (in Bits)
                                 ; besetzen
MULTI2:  ADD     HL,HL           ; Akkumulator verdoppeln
         RLCA                    ; hoechstes Bit des
                                 ; Multiplikators in
                                 ; Uebertrag-Flag bringen,
                                 ; gleichzeitig Multiplikator
                                 ; zirkulaer links rotieren
         JP      NC,NULL2        ; keine Addition erforderlich
         ADD     HL,DE           ; Multiplikand zu Akkumulator
                                 ; addieren
NULL2:   DJNZ    MULTI2          ; alle Bits des Multiplikators
```

```
                                    ; verarbeiten
        LD      DE,(XQUADR)  ; Quadrat der X-Koordinate holen
        ADD     HL,DE        ; Quadrat des Abstands berechnen
        EX      DE,HL        ; und sichern
        LD      HL,(ABSTND)  ; Quadrat des bisher groessten
                             ; Abstands holen
        SCF                  ; Abstaende vergleichen
        SBC     HL,DE
        JP      NC,WEITER    ; neuer Abstand kleiner als
                             ; alter Abstand
        LD      (PUNKT),IX   ; Zeiger auf neuen Punkt sichern
        LD      (ABSTND),DE  ; Quadrat des neuen Abstands
                             ; merken
WEITER: INC     IX           ; auf naechsten Punkt zeigen
        INC     IX
        DEC     C            ; Anzahl der restlichen Punkte
                             ; berechnen
        JP      NZ,PRUEFE    ; weiteren Punkt testen
        LD      IX,(PUNKT)   ; Zeiger auf weitest entfernten
                             ; Punkt holen
        LD      B,(IX+1)     ; X-Koordinate holen
        LD      C,(IX+2)     ; Y-Koordinate holen
        LD      DE,(ABSTND)  ; Quadrat des Abstands holen
```

Kapitel 17.2

1. Die gesamte Datenstruktur belegt 40 Bits = 5 Bytes. Da die Komponentengrenzen (außer am Anfang und Ende der Datenstruktur) niemals mit Bytegrenzen übereinstimmen, können wir in der Definition die Struktur nicht zum Ausdruck bringen; dies geschieht erst durch den Algorithmus. Die Vereinbarung der Datenstruktur lautet also:

```
ZEIT:   DEFS    5       ; 5 Bytes Speicherplatz fuer die
                        ; Komponenten des Verbunds
                        ; reservieren
```

Die Feinstruktur würde folgendermaßen aussehen:

Sekunde	Byte 0 / Bit 7 - Bit 2
Minute	Byte 0 / Bit 1 - Bit 0 und Byte 1 / Bit 7 - Bit 4
Stunde	Byte 1 / Bit 3 - Bit 0 und Byte 2 / Bit 7

Tag	Byte 2 / Bit 6 - Bit 2
Monat	Byte 2 / Bit 1 - Bit 0 und Byte 3 / Bit 7 - Bit 6
Jahr	Byte 3 / Bit 5 - Bit 0 und Byte 4 / Bit 7 - Bit 3
Wochentag	Byte 4 / Bit 2 - Bit 0

Wir müssen nun mit Techniken arbeiten, die wir im Kapitel »Bit-Manipulationen« kennengelernt haben. Die folgende Lösung ist weder speicherplatz- noch laufzeitoptimal, stellt aber eine Art von Standardlösung für gepackte Strukturen dar.

Wir legen einen Variablenblock an, in dem wir die einzelnen Komponenten des Verbunds ablegen können:

```
BASIS:
SEKUND:   DEFS    1          ; Sekunde
MINUTE:   DEFS    1          ; Minute
STUNDE:   DEFS    1          ; Stunde
TAG:      DEFS    1          ; Tag
MONAT:    DEFS    1          ; Monat
JAHR:     DEFS    2          ; Jahr
WOTAG:    DEFS    1          ; Wochentag
```

Als erstes »entpacken« wir den Verbund, das heißt, wir isolieren die einzelnen Komponenten und speichern sie im Variablenblock ab:

```
          LD      IX,ZEIT          ; Basis-Adresse des Verbunds
          LD      IY,BASIS         ; Basis-Adresse des
                                   ; Variablenblocks
          LD      C,(IX+4)         ; Hinterteil des Jahres und
                                   ; Wochentages
          LD      A,C              ; Hinterteil des Jahres und
                                   ; Wochentag kopieren
          AND     00000111B        ; Wochentag isolieren
          LD      (IY+WOTAG-BASIS),A    ; Wochentag abspeichern
          LD      H,(IX+0)         ; Sekunde und Vorderteil der
                                   ; Minute
          LD      L,(IX+1)         ; Hinterteil der Minute und
                                   ; Vorderteil der Stunde
          LD      A,(IX+2)         ; Hinterteil der Stunde, Tag und
                                   ; Vorderteil des Monats
          LD      B,(IX+3)         ; Hinterteil des Monats und
                                   ; Vorderteil des Jahres
          SRL     H                ; gesamten Verbund um 2 Bits
```

```
                                ; nach rechts schieben, um
                                ; Sekunde zu isolieren
RR          L
RRA
RR          B
RR          C
SRL         H
RR          L
RRA
RR          B
RR          C
LD          (IY+0),H            ; Sekunde abspeichern
LD          D,A                 ; Hinterteil von Stunde und
                                ; Tag kopieren
AND         00011111B           ; Tag isolieren
LD          (IY+TAG-BASIS),A        ; Tag abspeichern
SRL         L                   ; Verbundteil, bestehend aus
                                ; Minute und Stunde, um 2 Bits
                                ; nach rechts verschieben,
                                ; um Minute zu isolieren
RR          D
SRL         L
RR          D
LD          (IY+MINUTE-BASIS),L     ; Minute abspeichern
SRL         D                   ; Stunde um 3 Bits nach rechts
                                ; verschieben, um rechtsbuendig
                                ; zu machen
SRL         D
SRL         D
LD          (IY+STUNDE-BASIS),D     ; Stunde abspeichern
SRL         B                   ; Verbundteil, bestehend aus
                                ; Monat und Jahr, um ein Bit
                                ; nach rechts verschieben, um
                                ; Jahr rechtsbuendig zu machen
RR          C
LD          E,B                 ; Monat und Vorderteil von Jahr
                                ; kopieren
SRL         E                   ; Monat um 3 Bits nach rechts
                                ; schieben, um rechtsbuendig
                                ; zu machen
SRL         E
SRL         E
LD          (IY+MONAT-BASIS),E      ; Monat abspeichern
```

```
LD      A,B             ; Monat und Vorderteil von Jahr
                        ; kopieren
AND     00000111B       ; Vorderteil von Jahr isolieren
LD      B,A             ; Jahr im HL-Register
                        ; zusammensetzen
LD      (JAHR),BC       ; Jahr abspeichern
```

Wir inkrementieren nun zunächst die Sekunde und - falls nötig - die Minute und Stunde.

```
INC     (IY+0)          ; Sekunden inkrementieren
LD      A,59            ; groesster Wert der Sekunde
CP      H               ; mit altem Wert der Sekunde
                        ; vergleichen
JP      NZ,PACKEN       ; kein Sekundenueberlauf,
                        ; weiter mit Packen des Verbunds
LD      (IY+0),0        ; Sekunde ruecksetzen
INC     (IY+MINUTE-BASIS)       ; Minuten inkrementieren
CP      L               ; groessten Wert der Minute (59)
                        ; mit altem Wert der Minute
                        ; vergleichen
JP      NZ,PACKEN       ; kein Minutenueberlauf,
                        ; weiter mit Packen des Verbunds
LD      (IY+MINUTE-BASIS),0     ; Minute ruecksetzen
INC     (IY+STUNDE-BASIS)       ; Stunde inkrementieren
LD      A,23            ; groesster Wert der Stunde
CP      D               ; mit altem Wert der Stunde
                        ; vergleichen
JP      NZ,PACKEN       ; kein Stundenueberlauf,
                        ; weiter mit Packen des Verbunds
LD      (IY+STUNDE-BASIS),0     ; Stunde ruecksetzen
```

Nun folgt die Behandlung des Stundenüberlaufs, die in der Inkremetierung des Tags besteht. Zuerst wird der Wochentag inkrementiert:

```
INC     (IY+WOTAG-BASIS)        ; Wochentag
                                ; inkrementieren
LD      A,7             ; auf Ueberlauf des
                        ; Wochentags testen
CP      (IY+WOTAG-BASIS)
JP      NZ,INKTAG       ; kein Ueberlauf des Wochentags
LD      (IY+WOTAG-BASIS),0      ; Wochentag ruecksetzen
```

Als letztes müssen wir noch den Tag - und eventuell Monat und Jahr - inkrementieren. Die

Anzahl der Tage eines Monats entnehmen wir folgender Tabelle (für den Februar werden provisorisch 28 Tage angenommen); die Tabelle steht irgendwo vor oder nach dem gesamten Objekt-Code der Routine.

```
MONATE:  DEFB    31          ; Januar
         DEFB    28          ; Februar (provisorisch)
         DEFB    31          ; Maerz
         DEFB    30          ; April
         DEFB    31          ; Mai
         DEFB    30          ; Juni
         DEFB    31          ; Juli
         DEFB    31          ; August
         DEFB    30          ; September
         DEFB    31          ; Oktober
         DEFB    30          ; November
         DEFB    31          ; Dezember
```

Wir verschaffen uns die Anzahl der Tage des laufenden Monats (für Februar provisorisch 28 Tage) im D-Register:

```
INKTAG:  LD      HL,MONATE-1 ; Basis-Adresse der Tabelle,
                             ; bezogen auf den Index Null
         LD      D,0         ; Monat zu Relativadresse machen
         ADD     HL,DE       ; Adresse der Anzahl der Tage
                             ; des laufenden Monats berechnen
         LD      D,(HL)      ; Anzahl der Tage des laufenden
                             ; Monats beschaffen
         LD      A,(IY+TAG-BASIS)      ; alten Wert des Tags
                                       ; holen
         INC     (IY+TAG-BASIS)        ; Tag inkrementieren
         CP      D           ; mit groesstem Wert des Tags
                             ; vergleichen
         JP      C,PACKEN    ; kein Tagueberlauf,
                             ; weiter mit Packen des Verbunds
         LD      (IY+TAG-BASIS),1      ; Tag ruecksetzen
         LD      A,2         ; auf Monat Februar testen
         CP      E
         JP      Z,FEBRUA    ; Februar benoetigt
                             ; Sonderbehandlung wegen
                             ; der Schaltjahre
         INC     (IY+MONAT-BASIS)      ; Monat inkrementieren
         LD      A,12        ; groesster Wert von Monat
         CP      E           ; mit vergangenem Monat
```

```
                                    ; vergleichen
        JP      NZ,PACKEN           ; kein Monatueberlauf,
                                    ; weiter mit Packen des Verbunds
        LD      (IY+MONAT-BASIS),1  ; Monat ruecksetzen
        INC     BC                  ; Jahr inkrementieren
        LD      (JAHR),BC           ; Jahr abspeichern
        JP      PACKEN              ; weiter mit Packen des Verbunds
```

Beim Februar können nun noch drei Fälle vorliegen:

1. Der Tag hatte den Wert 29; in diesem Fall muß der Tag rückgesetzt und der Monat inkrementiert werden.
2. Der Tag hatte den Wert 28 und es liegt kein Schaltjahr vor; dann muß ebenfalls der Tag rückgesetzt und der Monat inkrementiert werden.
3. Der Tag hatte den Wert 28 und es liegt ein Schaltjahr vor; dann war das Inkrementieren des Tags zulässig und wir sind fertig.

```
FEBRUA: LD      A,29                ; feststellen, ob alter Wert
                                    ; des Tags 29 war
        CP      (IY+TAG-BASIS)
        JP      NC,NICH29           ; Tag hatte nicht den Wert 29,
INKMON: LD      (IY+TAG-BASIS),1    ; Tag ruecksetzen
        INC     (IY+MONAT-BASIS)    ; Monat inkrementieren
        JP      PACKEN              ; weiter mit Packen des Verbunds
```

Jetzt muß eine Prüfung auf Schaltjahr durchgeführt werden. Die Regeln dazu lauten:

1. Ist das Jahr nicht glatt durch 4 teilbar, so ist es kein Schaltjahr.
2. Ist das Jahr glatt durch 100 teilbar, nicht aber glatt durch 400 teilbar, so ist es kein Schaltjahr.
3. Alle übrigen Jahre sind Schaltjahre.

```
NICH29: LD      H,B                 ; pruefen, ob Jahr glatt durch 4
                                    ; teilbar ist, wenn nicht, dann
                                    ; Tag ruecksetzen und Monat
                                    ; inkrementieren
        LD      L,C
        SRL     H
        RR      L
        JP      C,INKMON
        SRL     H
        RR      L
        JP      C,INKMON
        XOR     A                   ; pruefen, ob Jahr glatt durch
```

```
                                        ; 100 teilbar ist, wenn nicht,
                                        ; dann liegt Schaltjahr vor
            DEC     A
            LD      DE,25
            ADD     HL,DE
SUBTRA:     INC     A
            SBC     HL,DE
            JP      C,PACKEN        ; Schaltjahr,
                                    ; weiter mit Packen des Verbunds
            JP      NZ,SUBTRA       ; Division durch Subtraktion
                                    ; weiterfuehren
            SRL     A               ; pruefen, ob Jahr glatt durch
                                    ; 400 teilbar ist, wenn nein,
                                    ; dann Tag ruecksetzen und
                                    ; Monat inkrementieren
            JP      C,INKMON
            SRL     A
            JP      C,INKMON
```

Als letzte Aktion setzen wir nun aus den Komponenten wieder einen gepackten Verbund zusammen:

```
PACKEN:     LD      HL,(JAHR)       ; Jahr holen
            LD      A,(MONAT)       ; Monat holen
            ADD     A,A             ; Monat um 3 Bits nach links
                                    ; schieben
            ADD     A,A
            ADD     A,A
            OR      H               ; Monat vor Vorderteil von Jahr
                                    ; einfuegen
            LD      H,A
            ADD     HL,HL           ; Verbund aus Monat und Jahr um
                                    ; ein Bit nach links schieben
            LD      A,(TAG)         ; Tag holen
            ADD     HL,HL           ; Verbund aus Tag, Monat und
                                    ; Jahr um 2 Bits nach links
                                    ; schieben
            ADC     A,A
            ADD     HL,HL
            ADC     A,A
            LD      E,(IY+STUNDE-BASIS)     ; Stunde holen
            ADD     A,A             ; Hinterteil von Stunde vor
                                    ; Verbund aus Tag und Vorderteil
```

```
                                    ; einfuegen
        SRL     E
        RRA
        LD      (IX+2),A            ; Hinterteil von Stunde, Tag und
                                    ; Vorderteil von Monat
                                    ; abspeichern
        LD      (IX+3),H            ; Hinterteil von Monat und
                                    ; Vorderteil von Jahr abspeichern
        LD      A,L                 ; Wochentag hinter Hinterteil von
                                    ; Jahr einfuegen
        OR      (IY+WOTAG-BASIS)
        LD      (IX+4),A            ; Hinterteil von Jahr und
                                    ; Wochentag abspeichern
        LD      A,(MINUTE)          ; Minute holen
        ADD     A,A                 ; Minute um 2 Bits nach links
                                    ; schieben
        ADD     A,A
        LD      D,(IY+SEKUND-BASIS) ; Sekunde holen
        ADD     A,A                 ; Vorderteil von Minute hinter
                                    ; Sekunde einfuegen
        RL      D
        ADD     A,A
        RL      D
        LD      (IX+0),D            ; Sekunde und Vorderteil von
                                    ; Minute abspeichern
        OR      E                   ; Vorderteil von Stunde hinter
                                    ; Hinterteil von Minute einfuegen
        LD      (IX+1),A            ; Hinterteil von Minute und
                                    ; Vorderteil von Stunde
                                    ; abspeichern
```

Kapitel 17.3

1. Die zugehörige Datenstruktur lautet:

```
BASIS:
DISKRI:   DEFS    1         ; Diskriminator, 0 steht fuer Zahl
                            ; 1 steht fuer zwei Zeichen
ZAHL:
ZEICH1:   DEFS    1         ; Zahl beziehungsweise
                            ; erstes Zeichen
ZEICH2:   DEFS    1         ; zweites Zeichen, unbenutzt bei
                            ; Speicherung einer Zahl
```

Das entsprechende Programm lautet:

```
        LD      IY,BASIS        ; Basis-Adresse des Verbunds
        LD      A,D             ; erstes Zeichen holen
        CP      '9'+1           ; auf Dezimalziffer testen
        JP      NC,KEINEZ       ; keine Dezimalziffer
        SUB     '0'             ; auf Dezimalziffer testen,
                                ; gegebenenfalls Ziffer in
                                ; Ziffernwert umrechnen
        JP      C,KEINEZ        ; keine Dezimalziffer
        LD      C,A             ; Ziffernwert sichern
        ADD     A,A             ; Ziffernwert verzehnfachen
        ADD     A,A
        ADD     A,C
        ADD     A,A
        LD      C,A             ; gewichteten Ziffernwert sichern
        LD      A,E             ; zweites Zeichen holen
        CP      '9'+1           ; auf Dezimalziffer testen
        JP      NC,KEINEZ       ; keine Dezimalziffer
        SUB     '0'             ; auf Dezimalziffer testen,
                                ; gegebenenfalls Ziffer in
                                ; Ziffernwert umrechnen
        JP      C,KEINEZ        ; keine Dezimalziffer
        ADD     A,C             ; Zahl berechnen
        LD      (IX+DISKRI-BASIS),0     ; Diskriminator mit
                                        ; Kennung fuer
                                        ; Zahl beschreiben
        LD      (IX+ZAHL-BASIS),A       ; Zahl abspeichern
        JP      FERTIG          ; Operation durchgefuehrt, zweite
                                ; Komponente bleibt ungenutzt
KEINEZ: LD      (IX+DISKRI-BASIS),1     ; Selektor mit Kennung
                                        ; fuer zwei Zeichen
                                        ; beschreiben
        LD      (IX+ZEICH1-BASIS),D     ; erstes Zeichen ablegen
        LD      (IX+ZEICH2-BASIS),E     ; zweites Zeichen ablegen
FERTIG: NOP                     ; gemeinsame Fortsetzungsstelle
```

2. Als Bestimmungsstücke für ein Dreieck wählen wir die Länge der drei Seiten (je ein Wort), für ein Quadrat die Seitenlänge (ein Wort), für ein Rechteck die Länge und die Breite (je ein Wort), für einen Kreis den Radius (ein Wort). Die Datenstruktur lautet damit:

```
BASIS:
DISKRI: DEFS    1               ; Diskriminator,
```

```
                                        ; 0 steht fuer Dreieck,
                                        ; 1 steht fuer Quadrat,
                                        ; 2 steht fuer Rechteck,
                                        ; 3 steht fuer Kreis
SEITE1:
KANTE:
LAENGE:
RADIUS:   DEFS      2                 ; erste Seite fuer Dreieck,
                                        ; Kantenlaenge fuer Quadrat,
                                        ; Laenge fuer Rechteck,
                                        ; Radius fuer Kreis
SEITE2:
BREITE:   DEFS      2                 ; zweite Seite fuer Dreieck,
                                        ; Breite fuer Rechteck,
                                        ; sonst ungenutzt
SEITE3:   DEFS      2                 ; dritte Seite fuer Dreieck,
                                        ; sonst ungenutzt
```

Kapitel 18.1

1. Ein Beispiel wäre:

```
                                        ; Programmbereich
          LD        SP,0                ; Stapel-Zeiger initialisieren
          PUSH      BC                  ; Registerinhalte sichern
          PUSH      DE
          PUSH      IX
          PUSH      IY
          LD        BC,0                ; Register ueberschreiben
          LD        DE,0
          D         IX,0
          LD        IY,0
          POP       IY                  ; Register restaurieren
          POP       IX
          POP       DE
          POP       BC
                                        ; Datenbereich
          ORG       10000H-512          ; Anfangsadresse
                                        ; des Speicherbereichs
                                        ; fuer den Stapel
STAPEL:   DEFS      512                 ; Speicherplatz fuer Stapel
                                        ; reservieren
```

Achte auf die Reihenfolge, in der die gesicherten Register restauriert werden.

2. Zur Durchführung der Operation genügt es, wenn der Stapel-Zeiger auf ein Wort im RAM weist.

```
EX      DE,HL
EX      (SP),HL
EX      (SP),IY
EX      (SP),HL
EX      DE,HL
```

3. Wir nehmen an, daß wir außer den beiden Stapeln, auf denen wir arbeiten, noch einen weiteren – den normalen Stapel des Z80 – besitzen und daß der Stapel-Zeiger gerade auf diesen weist.

```
PUSH    AF          ; Registerinhalte sichern
PUSH    BC
PUSH    DE
PUSH    HL
LD      HL,0        ; Stapel-Zeiger sichern
ADD     HL,SP
LD      SP,(SP2)    ; zweiten Stapel ansprechen
POP     BC          ; die obersten zwei Elemente
                    ; vom zweiten Stapel nehmen
POP     DE
LD      (SP2),SP    ; neuen Wert des Stapel-Zeigers
                    ; des zweiten Stapels sichern
LD      SP,(SP1)    ; ersten Stapel ansprechen
PUSH    DE          ; die beiden Elemente auf den
                    ; ersten Stapel werfen
PUSH    BC
LD      (SP1),SP    ; neuen Wert des Stapel-Zeigers
                    ; des ersten Stapels sichern
LD      SP,HL       ; normalen Stapel ansprechen
POP     HL          ; Register restaurieren
POP     DE
POP     BC
POP     AF
```

4. Das folgende Programm ist nicht optimal, sondern in Hinsicht auf möglichst große Verständlichkeit geschrieben. Wir nehmen an, daß die Werte ganze Zahlen in 2-Komplement-Darstellung sind. Die drei Werte bezeichnen wir durch X, Y und Z.

```
PUSH    BC          ; X sichern
PUSH    DE          ; Y sichern
```

```
        PUSH    HL          ; Z sichern
        OR      A           ; Uebertrag-Flag loeschen
        ADC     HL,BC       ; X und Z addieren
        JP      PO,KUEB1    ; kein Ueberlauf
        POP     HL          ; alte Werte restaurieren
        POP     DE
        POP     BC
        JP      FERTIG      ; weiter an gemeinsamer
                            ; Fortsetzungsstelle
KUEB1:  EX      DE,HL       ; Summe aus X und Z
                            ; ins DE-Register bringen,
                            ; Y ins HL-Register bringen
        OR      A           ; Uebertrag-Flag loeschen
        ADC     HL,BC       ; X und Y addieren
        P       PO,KUEB2    ; kein Ueberlauf
        POP     HL          ; alte Werte restaurieren
        POP     DE
        POP     BC
        JP      FERTIG      ; weiter an gemeinsamer
                            ; Fortsetzungsstelle
KUEB2:  POP     BC          ; Z ins BC-Register bringen
        EX      (SP),HL     ; Y ins HL-Register bringen,
                            ; Summe aus X und Y sichern
        PUSH    HL          ; Y wieder sichern
        OR      A           ; Uebertrag-Flag loeschen
        ADC     HL,BC       ; Y und Z addieren
        JP      PO,KUEB3    ; kein Ueberlauf
        POP     DE          ; Y restaurieren
        POP     HL          ; Summe aus X und Y vernichten
        LD      H,B         ; Z restaurieren
        LD      L,C
        POP     BC          ; X restaurieren
        JP      FERTIG      ; weiter an gemeinsamer
                            ; Fortsetzungsstelle
KUEB3:  LD      B,H         ; Summe aus Y und Z
                            ; ins BC-Register bringen
        LD      C,L
        POP     HL          ; Y beseitigen
        POP     HL          ; Summe aus X und Y
                            ; ins HL-Register bringen
        INC     SP          ; X beseitigen
        INC     SP
FERTIG: NOP                 ; gemeinsame Fortsetzungsstelle
```

Kapitel 18.2

1. Wir nehmen ein weiteres Doppelregister zu Hilfe:

```
POP     BC
DEC     SP
LD      A,C
```

2. Die Lösung könnte zum Beispiel lauten:

```
PUSH    DE
INC     SP
```

3. Ein Lösungsvorschlag:

```
LD      D,E
PUSH    DE
INC     SP
```

4. Die Lösung lautet:

```
POP     BC
DEC     SP
```

5. Die Lösung lautet:

```
DEC     SP
POP     HL
```

Kapitel 18.3

1. Wir nehmen an, daß die Längenangabe ein Byte umfaßt. Den Kopiervorgang führen wir mit dem LDIR-Befehl durch; anschließend muß der Stapel-Zeiger korrigiert werden.

```
EX      DE,HL       ; Zeiger tauschen
LD      HL,0        ; Zeiger auf Zeichenkette holen
ADD     HL,SP
LD      B,0         ; Laenge der Zeichenkette holen
LD      C,(HL)
INC     BC          ; die Laenge des zu kopierenden
                    ; Speicherbereichs ist wegen der
                    ; Laengenangabe um ein Byte
```

```
        LDIR                    ; laenger als die Zeichenkette
                                ; Zeichenkette einschliesslich
                                ; Laengenangabe kopieren
        LD      SP,HL           ; Stapel-Zeiger korrigieren
```

2. Wir transportieren die 29 Bytes der Datenstruktur mittels eines LDIR-Befehls als Datenblock auf den Stapel und korrigieren den Stapel-Zeiger entsprechend:

```
        LD      BC,29           ; Laenge des Datenblocks
        PUSH    IX              ; Anfangsadresse des Datenblocks
                                ; kopieren
        POP     DE
        LD      HL,0            ; Wert des Stapel-Zeigers holen
        ADD     HL,SP
        OR      A               ; neuen Stapel-Zeiger berechnen
        SBC     HL,BC
        LD      SP,HL           ; Stapel-Zeiger korrigieren
        EX      DE,HL           ; Zeiger tauschen
        LDIR                    ; Datenblock auf Stapel legen
```

3. Das entsprechende Programmstück lautet:

```
        LD      HL,62           ; Laenge der zu vernichtenden
                                ; Datenstruktur
        ADD     HL,SP           ; neuer Wert des Stapel-Zeigers
        LD      SP,HL           ; Stapel-Zeiger korrigieren
```

Kapitel 19.1

1. Das ASCII-Zeichen übergeben wir im A-Register; das Ergebnis steht ebenfalls wieder im A-Register.

```
ASCBIN: SUB     '0'             ; Korrektur fuer Dezimalziffer
                                ; durchfuehren
        CP      0AH             ; auf Dezimalziffer testen
        RET     C               ; Dezimalziffer
        SUB     'A'-0AH-'0'     ; Zusatzkorrektur fuer Hex-Ziffer
        RET
```

Wir nehmen nun an, daß die höherwertige Ziffer im B-Register codiert ist, die niederwertige im C-Register. Das Byte bauen wir im A-Register auf.

```
        LD      A,B     ; hoeherwertige Ziffer holen
        CALL    ASCBIN  ; in binaere Codierung umwandeln
        ADD     A,A     ; Ziffer in
        ADD     A,A     ; hoeherwertigen
        ADD     A,A     ; Nibble
        ADD     A,A     ; bringen
        LD      B,A     ; Nibble abspeichern
        LD      A,C     ; niederwertige Ziffer holen
        CALL    ASCBIN  ; in binaere Codierung umwandeln
        OR      B       ; beide Nibbles zusammensetzen
```

2. Unser erstes Unterprogramm leistet die Umwandlung einer im A-Register stehenden ASCII-Ziffer in ihre Binärdarstellung:

```
ASCBIN: SUB     '0'     ; Korrektur fuer Dezimalziffer
        RET
```

Das nächste Unterprogramm addiert den Wert des A-Registers auf das DE-Register (unseren Akkumulator) auf:

```
ADD:    PUSH    HL      ; Registerinhalt sichern
        LD      H,0     ; Summand zu
        LD      L,A     ; Wort erweitern
        ADD     HL,DE   ; Summand aufaddieren
        EX      DE,HL   ; Akkumulator sichern
        POP     HL      ; Register restaurieren
        RET
```

Das zweite Unterprogramm führt die Summenbildung für die beiden Nibbles durch, auf die das HL-Register zeigt:

```
ADDBYT: XOR     A       ; Akkumulator loeschen
        CALL    ADDNIB  ; Nibble holen und aufaddieren
ADDNIB: RRD             ; Nibble holen
        JP      ADD     ; Nibble aufaddieren
```

Wir nehmen nun an, daß das Feld stets eine gerade Anzahl von Nibbles enthält und an einer Bytegrenze beginnt. Wir entscheiden uns für ein Feld mit Deskriptor, der die Anzahl der Bytes im Feld angibt. Zu Beginn soll das HL-Register auf das Feld zeigen.

```
ADDKET: PUSH    AF      ; Register-
        PUSH    BC      ; inhalte
        PUSH    HL      ; sichern
```

```
        LD      DE,0        ; Akkumulator loeschen
        LD      B,(HL)      ; Anzahl der Bytes holen
        INC     B           ; Schleife abweisend
        JP      TEST        ; machen
HOLE:   INC     HL          ; auf naechstes Byte zeigen
        CALL    ADDBYT      ; beide Ziffern bearbeiten
TEST:   DJNZ    HOLE        ; gesamtes Feld bearbeiten
        POP     HL          ; alle
        POP     BC          ; Register
        POP     AF          ; restaurieren
        RET
```

3. Wir schreiben zunächst ein Unterprogramm, das in einer Zeichenkette eine Folge von Leerzeichen überliest, beginnend bei dem Zeichen, auf welches das HL-Register zeigt:

```
LEER:   LD      A,' '       ; Testzeichen laden
TEST:   CP      (HL)        ; auf Leerzeichen testen
        RET     NZ          ; kein Leerzeichen
        INC     HL          ; auf naechstes Zeichen zeigen
        JP      TEST        ; eventuell weitere Leerzeichen
                            ; ueberlesen
```

Als nächstes brauchen wir ein Unterprogramm, das feststellt, ob das HL-Register auf eine ASCII-codierte Dezimalziffer zeigt. Wenn ja, so soll das Übertrag-Flag gesetzt werden.

```
ZIFFER: LD      A,(HL)      ; Zeichen holen
        CP      '0'         ; auf Dezimalziffer testen
        CCF                 ; Uebertrag-Flag invertieren
        RET     NC          ; keine Dezimalziffer
        CP      '9'+1       ; auf Dezimalziffer testen
        RET
```

Darauf aufbauend konstruieren wir eine Funktion, die alle Ziffern ab der Stelle überliest, auf die das HL-Register zeigt:

```
LIESZ:  CALL    ZIFFER      ; auf Dezimalziffer testen
        RET     NC          ; keine Dezimalziffer
        INC     HL          ; Dezimalziffer ueberlesen
        JP      LIESZ       ; alle Dezimalziffern ueberlesen
```

Damit sind wir in der Lage, unser Unterprogramm so zu schreiben, daß das Null-Flag genau dann gesetzt wird, wenn das HL-Register auf eine Zeichenkette zeigt, welche eine ganze Dezimalzahl darstellt. Die Zeichenkette soll dabei durch eine Endemarkierung begrenzt werden.

```
ZAHL:   CALL    LEER        ; fuehrende Leerzeichen
                            ; ueberlesen
        LD      A,(HL)      ; naechstes Zeichen holen
        CP      '+'         ; mit Pluszeichen vergleichen
        JP      Z,VORZEI    ; Zahl hat Vorzeichen
        CP      '-'         ; mit Minuszeichen vergleichen
        JP      NZ,BETRAG   ; Zahl hat kein Vorzeichen
VORZEI: INC     HL          ; Vorzeichen ueberlesen
        CALL    LEER        ; Leerzeichen zwischen Vorzeichen
                            ; und Betrag ueberlesen
BETRAG: CALL    ZIFFER      ; auf Dezimalziffer testen
        JP      C,EINEZ     ; eine Dezimalziffer gefunden
        XOR     A           ; Akkumulator loeschen
        INC     A           ; Null-Flag loeschen
        RET
EINEZ:  INC     HL          ; fuehrende Ziffer ueberlesen
        CALL    LIESZ       ; alle weiteren Ziffern
                            ; ueberlesen
        CALL    LEER        ; folgende Leerzeichen ueberlesen
        LD      A,(HL)      ; naechstes Zeichen holen
        CP      ENDE        ; mit Endemarkierung vergleichen
        RET
```

Kapitel 19.2

1. Das Halbieren einer vorzeichenlosen ganzen Zahl erledigen wir durch Rechtsverschiebung um ein Bit. Problematisch ist dabei die Veränderung der Flags. Wir lösen das Problem, indem wir die Flags (und ein Hilfsregister) sichern, später restaurieren und erst dann den Wert des Ergebnisses ins A-Register bringen, ohne die Flags nochmals zu verändern:

```
HALB:   PUSH    BC          ; Registerinhalt sichern
        PUSH    AF          ; Flags sichern
        LD      B,A         ; Operand kopieren
        SRL     B           ; Operand halbieren
        POP     AF          ; Flags restaurieren
        LD      A,B         ; Ergebnis kopieren
        POP     BC          ; Register restaurieren
        RET
```

2. Das A-Register und die Flags werden auf jeden Fall verändert (es sei denn, sie hätten vorher bereits zufällig die resultierenden Werte). Wenn der Kopiervorgang tatsächlich stattfindet, so werden zusätzlich die beiden Zeiger HL und DE um die Länge des verschobenen Spei-

cherbereichs erhöht; außerdem werden die Speicherzellen, in die kopiert wird, überschrieben.

Kapitel 19.3

1. Ein Beispiel für die Lösung der Aufgabe wäre:
   Die Funktionsnummer soll folgendermaßen codiert sein:

1	Addition
2	Subtraktion
3	Absoluter Betrag
4	Negieren

Wir erwarten die Funktionsnummer im E-Register. Der erste (und eventuell einzige) Operand soll im A-Register stehen, der (eventuell vorhandene) zweite Operand im D-Register. Das Ergebnis soll im A-Register zurückgegeben werden.

```
ARITH:  DEC   E        ; Auf Funktionsnummer 1 testen
        JP    NZ,NEINS ; Funktionsnummer nicht 1
        ADD   A,D      ; Summe bilden
        RET
NEINS:  DEC   E        ; Auf Funktionsnummer 2 testen
        JP    NZ,NZWEI ; Funktionsnummer nicht 2
        SUB   D        ; Differenz bilden
        RET
NZWEI:  DEC   E        ; Auf Funktionsnummer 3 testen
        JP    NZ,NDREI ; Funktionsnummer nicht 3
        OR    A        ; Vorzeichen des Operanden testen
        RET   P        ; Vorzeichen positiv
NDREI:  NEG            ; Operand negieren
        RET
```

2. Der erste Operand der Vergleichsoperation soll im Superregister HL&DE stehen, der zweite Operand im Superregister IX&IY. Beide Operanden wollen wir als vorzeichenlose ganze Zahlen interpretieren. Das Ergebnis soll im A-Register zurückgeliefert werden und folgende Codierung besitzen:

1	<
2	=
3	>

```
TEST:   PUSH  BC       ; Registerinhalte
        PUSH  HL       ; sichern
```

```
        PUSH    IX      ; Inhalt des IX-Registers ins
        POP     BC      ; BC-Register kopieren
        OR      A       ; Uebertrag-Flag loeschen
        SBC     HL,BC   ; Vergleich der hoeherwertigen
                        ; 16 Bits durchfuehren
        JP      C,KLEIN ; 1. Operand < 2. Operand
        JP      NZ,GROESS ; 1. Operand > 2. Operand
        PUSH    IY      ; Inhalt des IY-Registers ins
        POP     HL      ; HL-Register kopieren
        OR      A       ; Uebertrag-Flag loeschen
        SBC     HL,DE   ; Vergleich der niederwertigen
                        ; 16 Bits durchfuehren
        JP      C,GROESS ; 1. Operand > 2. Operand
        JP      NZ,KLEIN ; 1. Operand < 2. Operand
        LD      A,2     ; Codierung fuer =
        POP     HL      ; Register
        POP     BC      ; restaurieren
        RET
KLEIN:  LD      A,1     ; Codierung fuer <
        POP     HL      ; Register
        POP     BC      ; restaurieren
        RET
GROESS: LD      A,3     ; Codierung fuer >
        POP     HL      ; Register
        POP     BC      ; restaurieren
        RET
```

Diese Lösung läßt sich noch geringfügig optimieren.

Kapitel 19.4

1. Wir nehmen folgende Belegung des Speichers an:

```
LAENGE: DEFS    1       ; Laenge der Zeichenkette
KETTE:  DEFS    256     ; Speicherplatz fuer Zeichenkette
```

Wir suchen dann nach dem Endezeichen ENDE und berechnen danach die Anzahl der überlesenen Zeichen, das ist die Länge der Zeichenkette:

```
LKETTE: PUSH    AF      ; Register-
        PUSH    BC      ; inhalte
        PUSH    HL      ; sichern
```

```
        LD      A,ENDE      ; Endemarkierung als Suchzeichen
                            ; laden
        LD      BC,256      ; Maximallaenge der Zeichenkette
        LD      HL,KETTE    ; Anfangsadresse des Textes
        CPIR                ; nach Endemarkierung suchen
        LD      A,C         ; Anzahl der ueberlesenen
        CPL                 ; Zeichen berechnen
        LD      (LAENGE),A  ; Laenge abspeichern
        POP     HL          ; alle
        POP     BC          ; Register
        POP     AF          ; restaurieren
        RET
```

2. Die beiden Zahlen sollen in folgendem Speicherbereich stehen:

```
OP1:    DEFS    8           ; erster 64-Bit-Operand
OP2:    DEFS    8           ; zweiter 64-Bit-Operand
```

Die Rückgabe des Ergebnisses soll wieder im A-Register stattfinden (gemischte Speicher/Register-Schnittstelle):

```
TEST:   PUSH    BC          ; Register-
        PUSH    DE          ; inhalte
        PUSH    HL          ; sichern
        LD      B,8         ; Laenge der Operanden in Bytes
        LD      DE,OP1      ; Anfangsadresse des ersten
                            ; Operanden
        LD      HL,OP2      ; Anfangsadresse des zweiten
                            ; Operanden
TESTB:  LD      A,(DE)      ; Byte des ersten Operanden holen
        CP      (HL)        ; mit Byte des zweiten
                            ; Operanden vergleichen
        JP      C,KLEIN     ; 1. Operand < 2. Operand
        JP      NZ,GROESS   ; 1. Operand > 2. Operand
        INC     DE          ; auf naechstes Byte des ersten
                            ; Operanden zeigen
        INC     HL          ; auf naechstes Byte des zweiten
                            ; Operanden zeigen
        DJNZ    TESTB       ; gegebenenfalls alle Bytes der
                            ; beiden Operanden testen
        LD      A,2         ; Code fuer =
        OP      HL          ; alle
        POP     DE          ; Register
```

```
            POP     BC          ; restaurieren
            RET
KLEIN:      LD      A,1         ; Code fuer <
            POP     HL          ; alle
            POP     DE          ; Register
            POP     BC          ; restaurieren
            RET
GROESS:     LD      A,3         ; Code fuer >
            POP     HL          ; alle
            POP     DE          ; Register
            POP     BC          ; restaurieren
            RET
```

Auch hier könnten wir noch einige kleinere Optimierungen durchführen.

Ein wesentlicher Unterschied zwischen dieser Lösung und der von Aufgabe 2 aus Unterkapitel 19.3 besteht darin, daß die einzelnen Bytes hier uniform behandelt werden; dies resultiert zwangsläufig aus der geringen Anzahl von Registern des Z80 (erzwungene Speicher-Schnittstelle).

Kapitel 19.5

1. Wir erwarten auf dem Stapel unterhalb der Rückkehradresse zunächst die Anzahl der beteiligten Teilmengen, sodann die Inzidenzvektoren der Teilmengen. Jeder Inzidenzvektor besteht aus einem Byte (erinnern Sie sich daran, daß die Vereinigung von Mengen, die durch Inzidenzvektoren repräsentiert werden, mittels des OR-Befehls durchgeführt wird).

   Das folgende Unterprogramm baut die Daten des Stapels ab und legt statt dessen das Ergebnis (wieder als Inzidenzvektor) dort ab. Es werden dabei einige Register zerstört.

```
VEREIN:     POP     IX          ; Rueckkehradresse holen
            LD      HL,0        ; Zeiger auf die
            ADD     HL,SP       ; Daten berechnen
            XOR     A           ; Akkumulator loeschen,
                                ; entspricht der leeren Menge
            LD      B,(HL)      ; Anzahl der Teilmengen holen
            INC     B           ; Schleife abweisend
            JP      TEST        ; machen
HOLEN:      INC     HL          ; auf naechsten Inzidenzvektor
                                ; zeigen
            OR      (HL)        ; Vereinigung durchfuehren
TEST:       DJNZ    HOLEN       ; Vereinigung aller
                                ; Teilmengen bilden
```

```
        LD      (HL),A      ; Ergebnis auf den Stapel bringen
        LD      SP,HL       ; Stapel-Zeiger neu einrichten
        JP      (IX)        ; Ruecksprung durchfuehren
```

Kapitel 19.6

1. Vergleiche die Dokumentationen mit den Beispielen im Text. Prüfe, ob stets alle fünf Angaben vollständig aufgeführt sind: Name, Funktion, Parameter, Ergebnis, Seiteneffekte.

Kapitel 19.7

1. Wichtig ist es, die beteiligten Register vor dem BDOS-Aufruf zu sichern. Zu Beginn soll das HL-Register auf die Zeichenkette zeigen, die eine Längenangabe von einem Byte trägt.

```
        LD      C,02H       ; Funktionsnummer fuer
                            ; Ausgabe auf Bildschirm
        LD      B,(HL)      ; Laenge der Zeichenkette holen
        INC     B           ; Schleife abweisend
        JP      TEST        ; machen
AUSGAB: INC     HL          ; auf naechstes Zeichen zeigen
        LD      E,(HL)      ; Zeichen holen
        PUSH    BC          ; Registerinhalte
        PUSH    HL          ; sichern
        CALL    BDOS        ; Zeichen ausgeben
        POP     HL          ; Register
        POP     BC          ; restaurieren
TEST:   DJNZ    AUSGAB      ; gesamte Zeichenkette ausgeben
```

2. Die Zeichenkette wird mit Endemarkierung im Speicher abgelegt; das HL-Register zeigt auf den dazu bestimmten Speicherbereich. Der Leseprozeß bricht ab, sobald die Endemarkierung ENDE gelesen wurde.

```
LIES:   LD      C,01H       ; Funktionsnummer fuer
                            ; Eingabe von der Tastatur
        PUSH    HL          ; Zeiger sichern
        CALL    BDOS        ; Zeichen von der Tastatur holen
        POP     HL          ; Zeiger restaurieren
        LD      (HL),A      ; Zeichen ablegen
        INC     HL          ; auf naechsten freien
                            ; Speicherplatz zeigen
```

```
        CP      ENDE        ; Zeichen mit Endemarkierung
                            ; vergleichen
        JP      NZ,LIES     ; Lesen, bis Endemarkierung
                            ; gelesen wurde
```

3. Eine Lösung des Problems lautet:

```
FNEIN   EQU     01H         ; Funktionscode fuer Eingabe
                            ; eines Zeichens von der
                            ; Tastatur
FNAUS   EQU     02H         ; Funktionscode fuer Ausgabe
                            ; eines Zeichens auf Bildschirm
        LD      C,FNEIN     ; Funktionscode fuer Eingabe
                            ; laden
        CALL    BDOS        ; Zeichen von Tastatur holen
        LD      B,A         ; Zeichen sichern
        AND     11110000B   ; niederwertigen Nibble
                            ; ausmaskieren
        RRCA                ; Nibble durch
        RRCA                ; viermaliges Rechtsrotieren
        RRCA                ; rechtsbuendig
        RRCA                ; machen
        LD      C,FNAUS     ; Funktionscode fuer Ausgabe
                            ; laden
        PUSH    BC          ; Zeichen und Funktionscode
                            ; sichern
        CALL    AUS         ; Nibble codieren und ausgeben
        POP     BC          ; Zeichen und Funktionscode holen
        LD      A,B         ; Zeichen holen
        AND     00001111B   ; hoeherwertigen Nibble
                            ; ausmaskieren
AUS:    CP      0AH         ; auf Dezimalziffer testen
        JP      C,DEZIMA    ; Dezimalziffer
        ADD     A,'A'-0AH-'0' ; Zusatzkorrektur fuer
                            ; Hex-Ziffer durchfuehren
DEZIMA: ADD     A,'0'       ; Korrektur fuer Dezimalziffer
                            ; durchfuehren
        LD      E,A         ; Ausgabezeichen als Parameter
                            ; des BDOS-Aufrufs bereitstellen
        JP      BDOS        ; Zeichen ausgeben
```

4. Wir geben bis zur Eingabe des ersten Zeichens permanent Leerzeichen aus. Nach der Ausgabe eines Zeichens wird geprüft, ob eine Eingabe ansteht. Wenn ja, so wird das Zeichen gelesen; es ersetzt das bisherige Ausgabezeichen.

```
FNEIN   EQU     01H     ; Funktionscode fuer Eingabe
                        ; eines Zeichens von der
                        ; Tastatur
FNAUS   EQU     02H     ; Funktionscode fuer Ausgabe
                        ; eines Zeichens auf Bildschirm
FNSTAT  EQU     0BH     ; Funktionsnummer fuer Abfragen
                        ; des Tastatur-Status
        LD      E,' '   ; vor Eingabe eines Zeichens nur
                        ; Leerzeichen ausgeben
AUS:    LD      C,FNAUS ; Funktionsnummer fuer Ausgabe
                        ; laden
        PUSH    DE      ; Ausgabezeichen sichern
        CALL    BDOS    ; Zeichen ausgeben
        LD      C,FNSTAT ; Funktionsnummer fuer Abfragen
                        ; des Status laden
        CALL    BDOS    ; Status abfragen
        POP     DE      ; Ausgabezeichen restaurieren
        OR      A       ; Status analysieren
        JP      Z,AUS   ; kein Zeichen anstehend
        LD      C,FNEIN ; Funktionsnummer fuer Eingabe
                        ; laden
        CALL    BDOS    ; neues Zeichen einlesen
        LD      E,A     ; Zeichen kopieren
        JP      AUS     ; weiter ausgeben
```

Kapitel 19.8

1. Wir denken uns die Zeichenkette durch ein Endezeichen ENDE abgeschlossen. Zu Beginn soll das HL-Register auf die Zeichenkette zeigen. Unser rekursives Unterprogramm, welches das Null-Flag setzt, falls die Zeichenkette einen korrekten Ausdruck darstellt, könnte zum Beispiel lauten:

```
TESTA:  LD      B,ENDE  ; Endemarkierung ist Begrenzer
                        ; fuer gesamten Ausdruck
AUSDRU: LD      A,(HL)  ; Zeichen holen
        CP      '('     ; auf oeffnende Klammer testen
        P       Z,KLAMM ; Klammerausdruck
        CP      '0'     ; auf Ziffer testen
        RET     C       ; keine Ziffer,
                        ; Ausdruck nicht korrekt,
                        ; Null-Flag ist geloescht
        CP      '9'+1   ; auf Ziffer testen
```

```
        JP      C,ZIFFER    ; Ziffer gefunden
        RET     NZ          ; Ausdruck nicht korrekt
        DEC     A           ; Null-Flag loeschen
        RET
ZIFFER: INC     HL          ; Ziffer ueberlesen
        JP      TEST        ; auf Endetest springen
KLAMM:  INC     HL          ; oeffnende Klammer ueberlesen
        PUSH    BC          ; bisherigen Begrenzer sichern
        LD      B,')'       ; neuer Begrenzer ist
                            ; schliessende Klammer
        CALL    AUSDRU      ; pruefen, ob ein Ausdruck folgt
        POP     BC          ; alten Begrenzer restaurieren
        RET     NZ          ; Ausdruck nicht korrekt
        INC     HL          ; schliessende Klammer ueberlesen
TEST:   LD      A,(HL)      ; naechstes Zeichen holen
        CP      B           ; mit Begrenzer vergleichen
        RET     Z           ; Ausdruck korrekt
        CP      '+'         ; mit Pluszeichen vergleichen
        JP      Z,OPERAT    ; Operator gefunden
        CP      '-'         ; mit Minuszeichen vergleichen
        RET     NZ          ; Ausdruck nicht korrekt
OPERAT: INC     HL          ; Operator ueberlesen
        JP      AUSDRU      ; Ausdruck weiter analysieren
```

Zum Verständnis des Programms: Jeder Ausdruck beginnt mit einer Ziffer oder einer Klammer!

Die Rekursivität von Summe und Differenz haben wir gleich in eine iterative Form gebracht.

Kapitel 19.9

1. Wir verzichten auf die Restaurierung der benutzten Register. Als oberstes Stapel-Element erwarten wir (unterhalb der Rückkehradresse) den ersten Operanden, darunter den zweiten Operanden. Nach der Rückkehr befindet sich über dem ersten Operanden die Summe, darüber die Differenz der beiden Operanden.

```
ARITH:  POP     IX          ; Rueckkehradresse holen
        POP     DE          ; ersten Operanden holen
        POP     BC          ; zweiten Operanden holen
        PUSH    BC          ; und wieder sichern
        PUSH    DE          ; ersten Operanden sichern
        LD      H,D         ; ersten Operanden
        LD      L,E         ; kopieren
```

```
        ADD     HL,BC       ; Summe bilden
        PUSH    HL          ; Summe ablegen
        EX      DE,HL       ; ersten Operanden holen
        OR      A           ; Uebertrag-Flag loeschen
        SBC     HL,BC       ; Differenz bilden
        PUSH    HL          ; Differenz ablegen
        JP      (IX)        ; Ruecksprung
```

Die hier gezeigte Methode unterscheidet sich insofern von der des Textes, daß sie für Programme, die sich gegenseitig unterbrechen (siehe Kapitel »Unterbrechungen«), nicht geeignet ist.

Kapitel 20.1

1. Wir fügen die Unterprogramme VOLL und LEER direkt in die Unterprogramme FUELLE und LEERE ein. Den Produzentenzeiger laden wir dabei ins DE-Register, das durch die Vergleichsoperationen nicht beeinflußt wird.

```
FUELLE: PUSH    HL          ; Registerinhalte
        PUSH    DE          ; sichern
        LD      DE,(PZEIG)  ; Produzenten-Zeiger holen
        LD      HL,ENDE+1   ; Testgroesse = ENDE + 1
        OR      A           ; auf vollen
        SBC     HL,DE       ; Puffer testen
        JP      Z,VOLL      ; Puffer voll
        LD      (DE),A      ; Zeichen ablegen
        INC     DE          ; auf naechstes Zeichen zeigen
        LD      (PZEIG),DE  ; Produzenten-Zeiger sichern
VOLL:   POP     DE          ; Register
        POP     HL          ; restaurieren
        RET
LEERE:  PUSH    HL          ; Registerinhalte
        PUSH    DE          ; sichern
        LD      DE,(PZEIG)  ; Produzenten-Zeiger holen
        LD      HL,ANFANG   ; Testgroesse = ANFANG
        OR      A           ; auf leeren
        SBC     HL,DE       ; Puffer testen
        JP      Z,LEER      ; Puffer leer
        LD      HL,(KZEIG)  ; Konsumenten-Zeiger holen
        LD      A,(HL)      ; Zeichen aus Puffer nehmen
        PUSH    AF          ; und zusammen mit Flags sichern
        INC     HL          ; auf naechstes Zeichen zeigen
```

```
        LD      (KZEIG),HL      ; Konsumenten-Zeiger sichern
        OR      A               ; pruefen, ob Puffer-Zeiger
        SBC     HL,DE           ; uebereinstimmen
        CALL    Z,INIT          ; Puffer geleert,
                                ; Puffer-Zeiger ruecksetzen
        POP     AF              ; alle
LEER:   POP     DE              ; Register
        POP     HL              ; restaurieren
        RET
```

Kapitel 20.2

1. Die Optimierungen werden ähnlich wie in der vorhergehenden Aufgabe durchgeführt. Ich verzichte deshalb auf eine ausgearbeitete Lösung.

Kapitel 20.3

1. Die beiden Voraussetzungen, daß
   a) der Ringpuffer 256 Bytes lang ist,
   b) das niederwertige Byte der Anfangsadresse des Puffers Null ist,
   erlauben, daß beim Manipulieren von Puffer-Zeigern stets nur das niederwertige Byte verändert werden muß, während das höherwertige Byte stets einen konstanten Wert besitzt. Dementsprechend merken wir uns nur noch das niederwertige Byte der benötigten Zeiger. Die Größe PUFFL muß nun konstant 256 sein; die Größe ENDE ist dadurch ebenfalls determiniert. Wir setzen die beiden Größen an den entsprechenden Stellen der Unterprogramme direkt ein. Der Puffer bekommt dadurch folgende Struktur:

```
FLAGS:  DEFS    1           ; Voll-Flag und Leer-Flag
PZEIG:  DEFS    1           ; niederwertiges Byte des
                            ; Produzenten-Zeigers
KZEIG:  DEFS    1           ; niederwertiges Byte des
                            ; Konsumenten-Zeigers
ANFANG: DEFS    256         ; Speicherplatz fuer Puffer
HBYTE   EQU     ANFANG/256  ; hoeherwertiges Byte der
                            ; Adresse ANFANG
```

An den Unterprogrammen VOLL, LEER, VSETZ, VLOES, LSETZ, LLOES ändert sich nichts. Das Unterprogramm INIT passen wir an die neue Situation an:

```
INIT:   PUSH    AF          ; Registerinhalt sichern
        CALL    LSETZ       ; Puffer als leer kennzeichnen
```

```
        CALL    VLOES       ; Puffer als nicht-voll
                            ; kennzeichnen
        XOR     A           ; Akkumulator loeschen
        LD      (PZEIG),A   ; Produzenten-Zeiger auf
                            ; ANFANG setzen
        LD      (KZEIG),A   ; Konsumenten-Zeiger auf
                            ; ANFANG setzen
        POP     AF          ; Register restaurieren
        RET
```

Die Unterprogramme PENDE und KENDE sind überflüssig, weil die zyklische Struktur des Ringpuffers durch die Voraussetzungen a) und b) bei Inkrement-Befehlen auf dem niederwertigen Byte der Puffer-Zeiger automatisch erzwungen wird.

Die Funktion des Unterprogramms GLEICH führen wir direkt in den Routinen FUELLE und LEERE durch; es muß dabei nur das niederwertige Byte der beiden Zeiger auf Gleichheit geprüft werden.

```
FUELLE: CALL    VOLL
        RET     Z
        PUSH    HL
        PUSH    DE
        LD      HL,PZEIG
        LD      D,HBYTE     ; Produzenten-Zeiger
        LD      E,(HL)      ; holen
        LD      (DE),A      ; Zeichen ablegen
        INC     (HL)
        CALL    LLOES
        PUSH    AF
        LD      A,(HL)
        INC     HL          ; auf Konsumenten-Zeiger zeigen
        CP      (HL)
        CALL    Z,VSETZ
        POP     AF
        POP     DE
        POP     HL
        RET
LEERE:  CALL    LEER
        RET     Z
        PUSH    HL
        PUSH    DE
        LD      HL,KZEIG
        LD      D,HBYTE     ; Konsumenten-Zeiger
        LD      E,(HL)      ; holen
```

```
        LD      A,(DE)
        CALL    VLOES
        PUSH    AF
        LD      A,(HL)
        DEC     HL              ; auf Produzenten-Zeiger zeigen
        CP      (HL)
        CALL    Z,LSETZ
        POP     AF
        POP     DE
        POP     HL
        RET
```

Die gezeigte Technik ist typisch für das Ausnutzen von »Regelmäßigkeiten« der Problemstellung; es wird dabei Information in den Algorithmen versteckt (Verlagerung von Informationsgehalt aus den Datenstrukturen in die Ablaufstrukturen).

Natürlich könnten wir – wie in den beiden vorhergehenden Aufgaben – durch Einsetzen der Unterprogramme VOLL und LEER in die Unterprogramme FUELLE und LEERE noch weitere Optimierungen durchführen.

2. Der Speicherplatz, dessen Adresse um eins kleiner ist als der Wert des Konsumenten-Zeigers (zyklisch im Adreßbereich des Puffers gerechnet), bleibt ungenutzt; dieser Speicherplatz enthält das letzte vom Konsumenten gelesene Zeichen (abgesehen von der Situation zu Beginn der Bearbeitung).

   Der Puffer ist genau dann leer, wenn die Werte von Konsumenten-Zeiger und Produzenten-Zeiger übereinstimmen. Der Puffer ist genau dann voll, wenn der Produzenten-Zeiger auf den unbenutzbaren Speicherplatz weist.

   Beim Füllen des Puffers muß nach der Ablage eines Zeichens nicht auf nun vollen Puffer getestet werden; ebenso muß beim Leeren nach Entnahme eines Zeichens nicht auf nun leeren Puffer getestet werden.

   Der Test auf leeren Puffer (vor Entnahme eines Zeichens) wird geringfügig komplizierter, da nun statt der Abfrage eines Flags der Vergleich zweier Zeiger notwendig ist.

   Beim Test auf vollen Puffer (vor Ablage eines Zeichens) sind eventuell zwei Zeigervergleiche durchzuführen, um dem zyklischen Umlaufen der Zeiger Rechnung zu tragen. Als Abhilfe bietet es sich an, beim Leeren des Puffers den alten Wert des Konsumenten-Zeigers als Zeiger auf den unbenutzbaren Speicherplatz aufzuheben; es genügt dann – wie beim Test auf leeren Puffer – ein Zeigervergleich.

Kapitel 21.1

1. Wir kennzeichnen das Ende der Tabelle durch ein Null-Byte:

```
KOORD:  DEFB    12
        DEFB    44
```

```
        DEFB    21
        DEFB    16
        DEFB    39
        DEFB    55
        DEFB    22
        DEFB    117
        DEFB    98
        DEFB    3
        DEFB    39
        DEFB    44
        DEFB    16
        DEFB    57
        DEFB    83
        DEFB    32
        DEFB    61
        DEFB    9
        DEFB    0           ; Ende-Markierung
```

2. Wir verwenden einen Deskriptor, der die Anzahl der Einträge angibt:

```
MENGE:  DEFB    8           ; Tabelle mit 8 Eintraegen
        DEFB    'G'
        DEFB    'J'
        DEFB    'E'
        DEFB    'f'
        DEFB    'n'
        DEFB    'R'
        DEFB    't'
        DEFB    'Z'
```

3. Es bietet sich wieder die Form einer Tabelle mit Deskriptor an:

```
RABATT: DEFB    5           ; Tabelle mit 5 Eintraegen
        DEFW    50          ; Abnahmemenge
        DEFB    2           ; Rabatt
        DEFW    100         ; Abnahmemenge
        DEFB    3           ; Rabatt
        DEFW    200         ; Abnahmemenge
        DEFB    5           ; Rabatt
        DEFW    500         ; Abnahmemenge
        DEFB    8           ; Rabatt
        DEFW    1000        ; Abnahmemenge
        DEFB    10          ; Rabatt
```

Kapitel 21.2

1. Wir legen eine Tabelle mit einem Deskriptor an, der die Anzahl der Einträge angibt. Die Namen legen wir als Zeichenketten der festen Länge 6 ab, wobei rechts gegebenenfalls mit Leerzeichen aufgefüllt wird:

```
QUIZ:   DEFB    6           ; Tabelle mit 6 Eintraegen
        DEFM    'Huber '    ; Name
        DEFB    12          ; Punktwert
        DEFM    'Meier '    ; Name
        DEFB    28          ; Punktwert
        DEFM    'Schulz'    ; Name
        DEFB    21          ; Punktwert
        DEFM    'Gruber'    ; Name
        DEFB    18          ; Punktwert
        DEFM    'Jung '     ; Name
        DEFB    14          ; Punktwert
        DEFM    'Weiss '    ; Name
        DEFB    21          ; Punktwert
```

Wir geben uns nun einen ab 0 gezählten Index im A-Register vor. Der Index wird zuerst auf seine Gültigkeit geprüft. Bei gültigem Index wird sodann der Anfangsbuchstabe des Namens ins C-Register, die erreichte Punktzahl ins B-Register gebracht:

```
        LD      HL,QUIZ     ; Zeiger auf Tabelle
        CP      (HL)        ; Gueltigkeit des Index pruefen
        JP      NC,FEHLER   ; ungueltiger Index
        INC     HL          ; auf ersten Tabelleneintrag
                            ; zeigen
        EX      DE,HL       ; Basis-Adresse sichern
        LD      H,0         ; Index zu
        LD      L,A         ; Wort erweitern
        LD      B,H         ; Index
        LD      C,L         ; kopieren
        ADD     HL,HL       ; Relativadresse
        ADD     HL,BC       ; als siebenfaches
        ADD     HL,HL       ; des Index
        ADD     HL,BC       ; berechnen
        ADD     HL,DE       ; Adresse des gesuchten Eintrags
                            ; berechnen
        LD      C,(HL)      ; Anfangsbuchstabe holen
        LD      DE,6        ; Laenge eines Namens
```

```
        ADD     HL,DE       ; auf Punktwert zeigen
        LD      B,(HL)      ; Punktwert holen
```

Wir hätten auch eines der Indexregister als Zeiger verwenden können.

2. Die Tabelleneinträge sind die Elemente des Puffers, also vom Typ »Wort«. Der Produzent verlängert die Tabelle, indem er neue Einträge anfügt. Der Konsument entnimmt jeweils den ersten Tabelleneintrag; sodann werden die restlichen Elemente nach vorne geschoben.

   Die aktuelle Anzahl von Puffer-Elementen halten wir im Deskriptor der Tabelle fest. Als maximale Anzahl von Elementen geben wir hier 16 vor. Im leeren Zustand sieht der Puffer dann folgendermaßen aus:

```
PUFFER: DEFB    0           ; aktuelle Anzahl der Elemente:
                            ; Puffer ist jetzt leer
        DEFS    16*2        ; Speicherplatz fuer maximal 16
                            ; Elemente mit je 2 Bytes
                            ; Platzbedarf reservieren
```

Eine Initialisierungsroutine für den Puffer würde einfach lauten:

```
INIT:   XOR     A           ; Akkumulator loeschen
        LD      (PUFFER),A  ; Puffer ruecksetzen
        RET
```

Der Produzent legt mit folgender Routine ein im BC-Register angeliefertes Wort im Puffer ab, falls noch Platz ist. Ist der Puffer voll (Puffer-Überlauf), so wird dies durch ein gesetztes Null-Flag angezeigt.

```
FUELLE: LD      HL,PUFFER   ; Zeiger auf Puffer holen
        LD      A,(HL)      ; aktuelle Anzahl der Elemente
                            ; holen
        CP      16          ; mit maximaler Anzahl
                            ; von Elementen vergleichen
        RET     Z           ; Fehler: Puffer-Ueberlauf
        INC     (HL)        ; es wird ein zusaetzliches
                            ; Element im Puffer abgelegt
        INC     HL          ; auf erstes Puffer-Element zeigen
        LD      D,0         ; Index zu
        LD      E,A         ; Wort erweitern
        ADD     HL,DE       ; Adresse des neuen
        ADD     HL,DE       ; Puffer-Elements berechnen
        LD      (HL),C      ; Wort im
        INC     HL          ; Puffer
```

```
        LD      (HL),B      ; ablegen
        INC     D           ; Null-Flag loeschen
        RET
```

Der Konsument bringt mit folgendem Unterprogramm das erste Element des Puffers ins BC-Register. Ist der Puffer leer (Puffer-Unterlauf), so wird das Null-Flag gesetzt.

```
LEERE:  LD      HL,PUFFER   ; Zeiger auf Puffer holen
        LD      A,(HL)      ; aktuelle Anzahl der Elemente
                            ; holen
        OR      A           ; auf leeren Puffer testen
        RET     Z           ; Fehler: Puffer-Unterlauf
        PUSH    AF          ; Null-Flag sichern
        DEC     (HL)        ; ein Element des Puffers wird
                            ; entfernt
        INC     HL          ; auf erstes Element des Puffers
                            ; zeigen
        LD      D,H         ; Basis-Adresse
        LD      E,L         ; kopieren
        LD      C,(HL)      ; erstes Element aus
        INC     HL          ; dem Puffer
        LD      B,(HL)      ; entnehmen
        INC     HL          ; gegebenenfalls auf zweites
                            ; Element des Puffers zeigen
        DEC     A           ; Anzahl der zu verschiebenden
                            ; Elemente des Puffers berechnen
        JP      Z,FERTIG    ; Puffer ist jetzt leer,
                            ; nichts zu verschieben
        PUSH    BC          ; entnommenes Element sichern
        ADD     A,A         ; Laenge des zu verschiebenden
                            ; Speicherbereichs berechnen
        LD      B,0         ; Laenge des zu verschiebenden
        LD      C,A         ; Speicherbereichs kopieren
        LDIR                ; restliche Elemente des Puffers
                            ; verschieben
        INC     B           ; Null-Flag loeschen
        POP     BC          ; entnommenes Element
                            ; wieder holen
FERTIG: POP     AF          ; Null-Flag restaurieren
        RET
```

Kapitel 21.3

1. Wir bauen eine Tabelle mit Endemarkierung auf; als Endezeichen verwenden wir einen Stern. Die Namen haben eine feste Länge von 7 Zeichen und werden gegebenenfalls rechts mit Leerzeichen aufgefüllt. Die Tabelle lautet somit:

```
ALTER:  DEFM    'Petra '    ; Name
        DEFB    35          ; Alter
        DEFM    'Hans '     ; Name
        DEFB    28          ; Alter
        DEFM    'Otto '     ; Name
        DEFB    27          ; Alter
        DEFM    'Hanna '    ; Name
        DEFB    29          ; Alter
        DEFM    'Klaus '    ; Name
        DEFB    22          ; Alter
        DEFM    'Inge '     ; Name
        DEFB    27          ; Alter
        DEFM    'Heinz '    ; Name
        DEFB    27          ; Alter
        DEFM    'Claudia'   ; Name
        DEFB    26          ; Alter
        DEFM    'Peter '    ; Name
        DEFB    22          ; Alter
        DEFB    '*'         ; Endezeichen
```

Das folgende Unterprogramm kopiert die Einträge der Personen, die jünger als das vorgegebene Alter sind, in eine neue Tabelle mit gleicher Struktur. Der Anfang dieser Tabelle wird durch das DE-Register bezeichnet. Das Vergleichsalter steht im B-Register.

```
        LD      HL,ALTER    ; Zeiger auf Tabelle holen
TEST:   LD      A,'*'       ; Endezeichen laden
        CP      (HL)        ; mit naechstem Zeichen
                            ; der Tabelle vergleichen
        JP      Z,FERTIG    ; Ende der Tabelle erreicht
        PUSH    HL          ; Zeiger auf Tabellenelement
                            ; sichern
        LD      A,B         ; Vergleichsalter sichern
        D       BC,7        ; Laenge eines Namens
        ADD     HL,BC       ; auf Alter zeigen
        LD      B,A         ; Vergleichsalter restaurieren
        LD      A,(HL)      ; Alter holen
        CP      B           ; und vergleichen
```

```
            JP          C,KOPIE     ; Alter kleiner als
                                    ; Vergleichsalter,
                                    ; Tabellenelement kopieren
            INC         SP          ; Zeiger auf Tabellenelement
            INC         SP          ; vom Stapel entfernen
            INC         HL          ; auf naechstes Tabellenelement
                                    ; zeigen
            JP          TEST        ; gesamte Tabelle bearbeiten
KOPIE:      POP         HL          ; Zeiger auf Tabellenelement
                                    ; holen
            LD          A,B         ; Vergleichsalter sichern
            LD          BC,8        ; Laenge eines Tabellenelements
            LDIR                    ; Tabellenelement kopieren
            LD          B,A         ; Vergleichsalter restaurieren
            JP          TEST        ; gesamte Tabelle bearbeiten
FERTIG:     LD          (DE),A      ; Endezeichen an neue Tabelle
                                    ; anfuegen
```

2. Wir legen eine Tabelle an, in welcher jeder Eintrag ein Byte belegt; die Bits 4 bis 7 werden dabei nicht benutzt. Die Tabelle hat einen Deskriptor, in dem die Anzahl der Tabelleneinträge steht. Das HL-Register zeigt auf den Anfang der Tabelle.

   Den acht auszuzählenden Merkmalen ordnen wir je ein Byte Speicher zu, in dem die Zählgröße steht. Diese Variablen adressieren wir durch ein Indexregister. Die Variablendefinition lautet zum Beispiel:

```
VARIAB:                             ; Basis-Adresse des Variablenblocks
MAENNL:     DEFB        0           ; Zaehler fuer Merkmal maennlich
WEIBLI:     DEFB        0           ; Zaehler fuer Merkmal weiblich
VERHEI:     DEFB        0           ; Zaehler fuer Merkmal verheiratet
LEDIG:      DEFB        0           ; Zaehler fuer Merkmal ledig
ARBEIT:     DEFB        0           ; Zaehler fuer Merkmal Arbeiter
ANGEST:     DEFB        0           ; Zaehler fuer Merkmal Angestellter
BEAMTE:     DEFB        0           ; Zaehler fuer Merkmal Beamter
SELBST:     DEFB        0           ; Zaehler fuer Merkmal selbständig
```

Die Auszählung geht nun folgendermaßen vor sich:

```
            LD          IX,VARIAB   ; Basis-Adresse
            LD          B,(HL)      ; Anzahl der Tabelleneintraege
ZAEHLE:     INC         HL          ; auf naechsten Tabelleneintrag
                                    ; zeigen
            BIT         0,(HL)      ; Merkmal maennlich/weiblich
                                    ; testen
```

```
        JP      Z,MAENN         ; maennlich
        INC     (IX+WEIBLI-VARIAB)      ; Zaehler fuer Merkmal
                                        ; weiblich erhoehen
        JP      BIT1            ; Bit 1 testen
MAENN:  INC     (IX+MAENNL-VARIAB)      ; Zaehler fuer Merkmal
                                        ; maennlich erhoehen
BIT1:   BIT     1,(HL)          ; Merkmal verheiratet/ledig
                                ; testen
        JP      Z,VERH          ; verheiratet
        INC     (IX+LEDIG-VARIAB)       ; Zaehler fuer Merkmal
                                        ; ledig erhoehen
        JP      BIT2            ; Bit 2 testen
VERH:   INC     (IX+VERHEI-VARIAB)      ; Zaehler fuer Merkmal
                                        ; verheiratet erhoehen
BIT2:   BIT     2,(HL)          ; Merkmal Arbeiter/Angestellter/
                                ; Beamter/selbstaendig testen
        JP      Z,ARBANG        ; Arbeiter/Angestellter
        BIT     3,(HL)          ; Merkmal Beamter/selbstaendig
                                ; testen
        JP      Z,BEAM          ; Beamter
        INC     (IX+SELBST-VARIAB)      ; Zaehler fuer Merkmal
                                        ; selbstaendig erhoehen
        JP      TEST            ; auf Ende der Tabelle testen
BEAM:   INC     (IX+BEAMTE-VARIAB)      ; Zaehler fuer Merkmal
                                        ; Beamter erhoehen
        JP      TEST            ; auf Ende der Tabelle testen
ARBANG: BIT     3,(HL)          ; Merkmal Arbeiter/Angestellter
                                ; testen
        JP      Z,ARB           ; Arbeiter
        INC     (IX+ANGEST-VARIAB)      ; Zaehler fuer Merkmal
                                        ; Angestellter erhoehen
        JP      TEST            ; auf Ende der Tabelle testen
ARB:    INC     (IX+ARBEIT-VARIAB)      ; Zaehler fuer Merkmal
                                        ; Arbeiter erhoehen
TEST:   DJNZ    ZAEHLE          ; auf Ende der Tabelle testen,
                                ; gegebenenfalls Zaehlung
                                ; fortsetzen
```

3. Wir entfernen das Element, das im A-Register steht. Das HL-Register zeigt auf die Menge.

```
HERAUS: LD      B,(HL)          ; Anzahl der Elemente der Menge
                                ; holen
        INC     B               ; auf leere
```

```
            DEC     B       ; Menge testen
            RET     Z       ; leere Menge, nichts zu tun
            PUSH    HL      ; Zeiger auf Anzahl der Elemente
                            ; sichern
SUCHE:      INC     HL      ; auf naechstes Element zeigen
            CP      (HL)    ; mit zu entfernendem Element
                            ; vergleichen
            JP      Z,GEFUND ; zu entfernendes Element
                            ; gefunden
            DJNZ    SUCHE   ; gegebenenfalls gesamte Menge
                            ; durchsuchen
            POP     HL      ; Zeiger auf Menge restaurieren
            RET
GEFUND:     DEC     B       ; Anzahl der noch folgenden
                            ; Elemente berechnen
            JP      Z,FERTIG ; keine Elemente zu verschieben
OPIE:       INC     HL      ; auf naechstes Element zeigen
            LD      A,(HL)  ; naechstes Element holen
            DEC     HL      ; auf freien Platz zeigen
            LD      (HL),A  ; Element verschieben
            INC     HL      ; auf jetzt freien Platz zeigen
            DJNZ    KOPIE   ; alle folgenden Elemente
                            ; verschieben
FERTIG:     POP     HL      ; Zeiger auf Anzahl der Elemente
                            ; holen
            DEC     (HL)    ; ein Element wurde entfernt
            RET
```

Darauf aufbauend stellen wir die Mengendifferenz dar. Wir entnehmen aus der ersten Menge, auf die das HL-Register zeigt, der Reihe nach alle Elemente der zweiten Menge, auf die das DE-Register zeigt:

```
DIFFER:     LD      A,(DE)  ; Anzahl der Elemente der
                            ; zweiten Menge holen
            OR      A       ; auf leere Menge testen
            RET     Z       ; leere Menge, nichts zu tun
            LD      C,A     ; Anzahl der Elemente in
                            ; Zaehlgroesse laden
ENTFER:     INC     DE      ; auf naechstes Element der
                            ; zweiten Menge zeigen
            LD      A,(DE)  ; naechstes Element der zweiten
                            ; Menge holen
            CALL    HERAUS  ; Element aus erster Menge
```

```
                                    ; entfernen
        DEC         C               ; Anzahl der restlichen Elemente
                                    ; der zweiten Menge berechnen
        JP          NZ,ENTFER       ; zweite Menge vollstaendig
                                    ; bearbeiten
        RET
```

Kapitel 22.2

1. Wir legen eine Tabelle für die Alternative an:

```
SPRUNG: DEFB        3               ; Tabelle mit 3 Eintraegen
        DEFB        '<'             ; erster Vergleichs-Wert
        DEFW        KLEIN           ; zugehoerige Sprungadresse
        DEFB        '='             ; zweiter Vergleichs-Wert
        DEFB        GLEICH          ; zugehoerige Sprungadresse
        DEFB        '>'             ; dritter Vergleichs-Wert
        DEFW        GROESS          ; zugehoerige Sprungadresse
```

Die drei vorkommenden Sprungadressen gehören zu folgenden Unterprogrammen:

```
KLEIN:  LD          A,B             ; linken Operanden laden
        CP          C               ; mit rechtem Operanden
                                    ; vergleichen
        RET
GEICH:  LD          A,B             ; linken Operanden laden
        CP          C               ; mit rechtem Operanden
                                    ; vergleichen
        SCF                         ; Uebertrag-Flag setzen
        RET         Z               ; Operanden stimmen ueberein
        CCF                         ; Uebertrag-Flag loeschen
        RET
GROESS: LD          A,C             ; rechten Operanden laden
        CP          B               ; mit linkem Operanden
                                    ; vergleichen
        RET
```

Der Aufruf erfolgt einfach durch

```
        LD          HL,SPRUNG       ; Adresse der Tabelle laden
        CALL        ALTERN          ; gewuenschtes Unterprogramm
                                    ; ausfuehren
```

2. Wir erweitern zunächst die Sprungtabelle um die Adresse der Fehlerroutine:

```
SPRUNG:  DEFB    3          ; Tabelle mit 3 Eintraegen
         DEFB    '<'        ; erster Vergleichs-Wert
         DEFW    KLEIN      ; zugehoerige Sprungadresse
         DEFB    '='        ; zweiter Vergleichs-Wert
         DEFB    GLEICH     ; zugehoerige Sprungadresse
         DEFB    '>'        ; dritter Vergleichs-Wert
         DEFW    GROESS     ; zugehoerige Sprungadresse
         DEFW    FEHLER     ; Adresse der Fehlerroutine
```

In der Fehlerroutine wird nur das Null-Flag geloescht:

```
FEHLER:  XOR     A          ; Akkumulator loeschen
         INC     A          ; Null-Flag loeschen
         RET
```

In den drei anderen Unterprogrammen muß das Null-Flag gesetzt werden, ohne daß das Übertrag-Flag verändert wird. Wir fügen jeweils das Programmstück

```
         LD      A,1        ; Akkumulator mit 1 laden
         DEC     A          ; Null-Flag setzen
```

ein, das den gewünschten Effekt hat:

```
KLEIN:   LD      A,B        ; linken Operanden laden
         CP      C          ; mit rechtem Operanden
                            ; vergleichen
         LD      A,1        ; Akkumulator mit 1 laden
         DEC     A          ; Null-Flag setzen
         RET
GLEICH:  LD      A,B        ; linken Operanden laden
         CP      C          ; mit rechtem Operanden
                            ; vergleichen
         SCF                ; Uebertrag-Flag setzen
         RET     Z          ; Operanden stimmen ueberein
         CCF                ; Uebertrag-Flag loeschen
         LD      A,1        ; Akkumulator mit 1 laden
         DEC     A          ; Null-Flag setzen
         RET
GROESS:  LD      A,C        ; rechten Operanden laden
         CP      B          ; mit linkem Operanden
                            ; vergleichen
```

```
        LD      A,1         ; Akkumulator mit 1 laden
        DEC     A           ; Null-Flag setzen
        RET
```

Die drei Unterprogramme können noch dadurch optimiert werden, daß das Programmstück nur in einem Unterprogramm auftaucht, in den beiden übrigen Fällen dagegen nur angesprungen wird:

```
KLEIN:  LD      A,B         ; linken Operanden laden
        CP      C           ; mit rechtem Operanden
                            ; vergleichen
NULL:   LD      A,1         ; Akkumulator mit 1 laden
        DEC     A           ; Null-Flag setzen
        RET
GLEICH: LD      A,B         ; linken Operanden laden
        CP      C           ; mit rechtem Operanden
                            ; vergleichen
        SCF                 ; Uebertrag-Flag setzen
        RET     Z           ; Operanden stimmen ueberein
        CCF                 ; Uebertrag-Flag loeschen
        JP      NULL        ; Null-Flag setzen
GROESS: LD      A,C         ; rechten Operanden laden
        CP      B           ; mit linkem Operanden
                            ; vergleichen
        JP      NULL        ; Null-Flag setzen
```

Kapitel 22.3

1. Es fehlen nur noch die Unterprogramme zur Prüfung der Attribute und die Unterprogramme zur Ausführung der jeweils zugeordneten Aktionen. Wir verwenden ein Indexregister, um die auf dem Stapel befindliche Zeilennummer zu adressieren. Die vier Unterprogramme zur Prüfung der Attribute lauten:

```
AUFOB:  CP      7           ; Codes fuer aufwaerts
                            ; sind: 7, 8, 9
        CCF                 ; Uebertrag-Flag invertieren
        RET     NC          ; keine Bewegung aufwaerts
                            ; gewuenscht
        LD      IX,0        ; Stapel-Zeiger ins
        ADD     IX,SP       ; Indexregister bringen
        LD      C,A         ; Code sichern
        LD      A,(IX+5)    ; aktuelle Zeilennummer
```

```
            CP      OBEN+1      ; kleinste Zeilennummer + 1
            LD      A,C         ; Code restaurieren
            RET
AUFNOB:     CP      7           ; Codes fuer aufwaerts
                                ; sind: 7, 8, 9
            CCF                 ; Uebertrag-Flag invertieren
            RET     NC          ; keine Bewegung aufwaerts
                                ; gewuenscht
            LD      IX,0        ; Stapel-Zeiger ins
            ADD     IX,SP       ; Indexregister bringen
            LD      C,A         ; Code sichern
            LD      A,OBEN      ; kleinste Zeilennummer
            CP      (IX+5)      ; aktuelle Zeilennummer
            LD      A,C         ; Code restaurieren
            RET
ABUNT:      CP      4           ; Codes fuer abwaerts
                                ; sind: 1, 2, 3
            RET     NC          ; keine Bewegung abwaerts
                                ; gewuenscht
            LD      IX,0        ; Stapel-Zeiger ins
            ADD     IX,SP       ; Indexregister bringen
            LD      C,A         ; Code sichern
            LD      A,UNTEN-1   ; groesste Zeilennummer - 1
            CP      (IX+5)      ; aktuelle Zeilennummer
            LD      A,C         ; Code restaurieren
            RET
ABNUNT:     CP      4           ; Codes fuer abwaerts
                                ; sind: 1, 2, 3
            RET     NC          ; keine Bewegung abwaerts
                                ; gewuenscht
            LD      IX,0        ; Stapel-Zeiger ins
            ADD     IX,SP       ; Indexregister bringen
            LD      C,A         ; Code sichern
            LD      A,(IX+5)    ; aktuelle Zeilennummer
            CP      UNTEN       ; groesste Zeilennummer
            LD      A,C         ; Code restaurieren
            RET
```

Um die Routinen AUFZEI, ABZEI und PIEPS ausformulieren zu können, müßten wir Informationen über die Hardware besitzen. Wir stützen uns deshalb auf eine Ausgaberoutine AUS, die im A-Register ein Zeichen erhält und dieses auf den Bildschirm ausgibt:

```
AUFZEI:     DEC     B           ; aktuelle Zeilennummer um
```

```
                                    ; eins reduzieren
        LD      A,AUF       ; Steuercode fuer aufwaerts
        JP      AUS         ; ausgeben
ABZEI:  INC     B           ; aktuelle Zeilennummer um
                            ; eins erhoehen
        LD      A,AB        ; Steuercode fuer abwaerts
        JP      AUS         ; ausgeben
PIEPS:  LD      A,GLOCKE    ; Steuercode fuer Piepsen
        JP      AUS         ; ausgeben
```

Für das Zeichen GLOCKE wird normalerweise der Code 07H (ASCII: bell) verwendet, für das Zeichen AB der Code 0AH (ASCII: line feed). Der Code für das Zeichen AUF und die Ausführung der Routine AUS sind dagegen stark hardwareabhängig.

Kapitel 23.1

1. Der Zeiger auf die erste Liste soll im HL-Register stehen, der Zeiger auf die Liste, die angehängt wird, im DE-Register. Den Zeiger auf die Gesamtliste liefern wir im HL-Register zurück.

   Wir haben zwei Fälle zu unterscheiden: Ist die erste Liste leer, so liefern wir einfach den Zeiger auf die zweite Liste zurück; ansonsten tragen wir den Zeiger auf die zweite Liste im letzten Element der ersten Liste als Nachfolger ein und liefern den Zeiger auf die erste Liste zurück.

```
        CALL    ISTNIL      ; feststellen, ob erste Liste
                            ; leer ist
        EX      DE,HL       ; Zeiger tauschen
        RET     Z           ; erste Liste leer
        EX      DE,H        ; Zeiger tauschen
        PUSH    HL          ; Zeiger auf erste Liste sichern
        PUSH    DE          ; Zeiger auf zweite Liste sichern
SUCHEN: CALL    NACHFO      ; feststellen, ob erste Liste zu
                            ; Ende, gegebenenfalls Zeiger auf
                            ; den Nachfolger berechnen
        EX      DE,HL       ; Zeiger tauschen
        JP      NZ,SUCHEN   ; weiter nach dem Ende der
                            ; ersten Liste suchen
        POP     DE          ; Zeiger auf zweite Liste holen
        INC     HL          ; auf Adresse des Nachfolgers
                            ; des letzten Elements der
                            ; ersten Liste zeigen
        LD      (HL),E      ; zweite Liste
```

```
        INC     HL          ; an erste Liste
        LD      (HL),D      ; anfuegen
        POP     HL          ; Zeiger auf erste Liste holen
        RET
```

2. Wenn die Werte der Elemente der Liste je »L« Bytes belegen, so muß zum Überlesen des Werts »L«-mal (statt einmal) inkrementiert werden; entsprechend oft muß beim Restaurieren dekrementiert werden.
   Falls »L« > 4, so verwendet man mit Vorteil folgendes Verfahren:

```
NACHFO: CALL    ISTNIL      ; Zeiger auf NIL testen
        RET     Z           ; ungueltiger Zeiger
        PUSH    HL          ; Zeiger sichern
        LD      DE,LAENGE   ; auf Adresse des
        ADD     HL,DE       ; Nachfolgers zeigen
        LD      E,(HL)      ; Adresse des
        INC     HL          ; Nachfolgers
        LD      D,(HL)      ; holen
        POP     HL          ; Zeiger restaurieren
        RET
```

3. Wenn wir vereinbaren, daß eine zyklisch verkettete lineare Liste stets mindestens ein Element enthält, so lautet das Unterprogramm:

```
NACHFO: INC     HL          ; auf Adresse des
                            ; Nachfolgers zeigen
        LD      E,(HL)      ; Adresse des
        INC     HL          ; Nachfolgers
        LD      D,(HL)      ; holen
        DEC     HL          ; Zeiger
        DEC     HL          ; restaurieren
        RET
```

Darf die zyklisch verkettete lineare Liste dagegen auch leer sein, so kann das Unterprogramm NACHFO aus dem Text unverändert übernommen werden (das Unterprogramm merkt sozusagen nicht, daß die Liste zyklisch ist).

Kapitel 23.2

Wir wollen stets annehmen, daß alle Mengen als einfach verkettete lineare Listen dargestellt sind; die Mengen sollen durch aufsteigend geordnete Repräsentationen von Zeichen dargestellt sein, so daß wir die Unterprogramme aus dem Text verwenden können.

1. Wir verknüpfen die beiden Listen zu einer Liste, welche die Vereinigungsmenge darstellt. Dazu benutzen wir das Unterprogramm HINEIN (diese Lösung ist allerdings nicht optimal). HL- und DE-Register zeigen zu Beginn auf die beiden Mengen; im HL-Register wird anschließend ein Zeiger auf die Vereinigungsmenge zurückgeliefert.

```
VEREIN:   PUSH    HL          ; Zeiger auf erste Liste sichern
          EX      DE,HL       ; Zeiger tauschen
          CALL    NACHFO      ; auf Ende der zweiten Liste
                              ; testen, gegebenenfalls
                              ; Nachfolger berechnen
          JP      Z,FERTIG    ; zweite Liste zu Ende
          EX      DE,HL       ; Zeiger tauschen
          EX      (SP),HL     ; Zeiger auf erste Liste holen,
                              ; Zeiger auf zweite Liste sichern
          LD      A,(DE)      ; Zeichen holen
          CALL    HINEIN      ; Element aus zweiter Liste in
                              ; erste Liste einfuegen
          POP     DE          ; Zeiger auf zweite Liste holen
          JP      VEREIN      ; weiter vereinigen
FERTIG:   POP     HL          ; Zeiger auf erste Liste holen
          RET
```

2. Wir versuchen, jedes Element der zweiten Menge aus der ersten Menge zu entnehmen; dazu verwenden wir das Unterprogramm LOESCH. Wieder zeigt HL auf die erste Menge, DE auf die zweite Menge; nach Abschluß der Operation befindet sich in HL ein Zeiger auf die Differenzmenge.

```
DIFFER:   PUSH    HL          ; Zeiger auf erste Liste sichern
          EX      DE,HL       ; Zeiger tauschen
          CALL    NACHFO      ; auf Ende der zweiten Liste
                              ; testen, gegebenenfalls
                              ; Nachfolger berechnen
          JP      Z,FERTIG    ; Ende der zweiten Liste
          LD      A,(HL)      ; Wert des Elements holen
          POP     HL          ; Zeiger auf erste Liste holen
          PUSH    DE          ; Zeiger auf zweite Liste sichern
          CALL    LOESCH      ; Element entfernen
          POP     DE          ; Zeiger auf zweite Liste holen
          JP      DIFFER      ; weiter Differenz bilden
FERTIG:   POP     HL          ; Zeiger auf erste Liste holen
          RET
```

3. Wir lösen das Problem, indem wir aus der ersten Menge diejenigen Elemente entfernen, die

nicht in der zweiten Menge vorkommen. Der Ablauf ist sehr ähnlich wie bei der vorhergehenden Aufgabe; wir verzichten deshalb hier auf eine ausgearbeitete Lösung.

4. Wir schreiben ein Unterprogramm, welches das Null-Flag setzt, falls die erste Menge (auf die das HL-Register zeigt) eine Teilmenge der zweiten Menge (auf die das DE-Register zeigt) ist.

```
TEILM:  LD      A,(HL)      ; Wert des ersten Elements
                            ; der ersten Liste holen
        PUSH    DE          ; Zeiger auf zweite Liste sichern
        CALL    NACHFO      ; auf Ende der ersten Liste
                            ; testen, gegebenenfalls
                            ; Nachfolger berechnen
        POP     HL          ; Zeiger auf zweite Liste holen
        RET     Z           ; erste Liste zu Ende,
                            ; erste Menge ist Teilmenge
                            ; der zweiten Menge
        CALL    ISTIN       ; testen, ob Element in
                            ; zweiter Liste vorkommt
        RET     NZ          ; Element nicht enthalten,
                            ; erste Menge keine Teilmenge
                            ; der zweiten Menge
        EX      DE,HL       ; Zeiger tauschen
        JP      TEILM       ; Rest der ersten Menge testen
```

Kapitel 23.3

1. Ersetzen von Arithmetik auf Zeigern durch Inkrementieren und Dekrementieren führt in diesem Fall zu optimierten Programmen.

Kapitel 23.4

1. Wir ersetzen – wie in der vorhergehenden Aufgabe – die Arithmetik auf den Zeigern durch Inkrementieren und Dekrementieren.

Kapitel 23.5

1. Zwei Bäume stimmen überein, wenn die Werte ihrer Wurzeln übereinstimmen, wenn die beiden linken Teilbäume übereinstimmen und wenn die beiden rechten Teilbäume übereinstimmen (rekursives Testverfahren). Wir gehen dabei also davon aus, daß die lineare Ordnung der Teilbäume eines Baums von Bedeutung ist.

Wir übergeben im HL- und DE-Register Zeiger auf die Wurzeln der beiden Bäume. Der Vergleichsalgorithmus setzt das Null-Flag, wenn beide Bäume übereinstimmen. Das Programm lautet:

```
VERGL:  CALL    ISTNIL      ; auf leeren Baum testen
        EX      DE,HL       ; Zeiger tauschen
        JP      NZ,NLEER1   ; erster Baum nicht leer
        CALL    ISTNIL      ; auf leeren Baum testen
        EX      DE,HL       ; Zeiger tauschen
        RET                 ; Baeume genau dann gleich,
                            ; wenn beide Baeume leer
NLEER1: CALL    ISTNIL      ; auf leeren Baum testen
        EX      DE,HL       ; Zeiger tauschen
        JP      NZ,NLEER2   ; zweiter Baum nicht leer
        XOR     A           ; Null-Flag loeschen
        INC     A
        RET                 ; Baeume sind nicht gleich
NLEER2: LD      A,(DE)      ; Wert der Wurzel des zweiten
                            ; Baums holen
        CP      (HL)        ; mit Wert der Wurzel des ersten
                            ; Baums vergleichen
        RET     NZ          ; Werte sind verschieden, also
                            ; sind die Baeume verschieden
        CALL    TBAUM       ; Test auf Gleichheit der beiden
                            ; linken Teilbaeume
        RET     NZ          ; die linken Teilbaeume sind
                            ; verschieden, also sind die
                            ; Baeume verschieden
TBAUM:  INC     HL          ; auf Adresse des Teilbaums des
                            ; ersten Baums zeigen
        LD      C,(HL)      ; Zeiger auf
        INC     HL          ; Teilbaum des ersten
        LD      B,(HL)      ; Baums holen
        PUSH    HL          ; Zeiger auf ersten Baum sichern
        EX      DE,HL       ; Zeiger tauschen
        INC     HL          ; auf Adresse des Teilbaums des
                            ; zweiten Baums zeigen
        LD      E,(HL)      ; Zeiger auf
        INC     HL          ; Teilbaum des zweiten
        LD      D,(HL)      ; Baums holen
        PUSH    HL          ; Zeiger auf zweiten Baum sichern
        EX      DE,HL       ; Zeiger tauschen
        PUSH    HL          ; Zeiger auf Teilbaum des
```

```
                                        ; zweiten Baums sichern
        OR      A           ; Uebertrag-Flag loeschen
        SBC     HL,BC       ; Zeiger auf die Teilbaeume
                            ; vergleichen
        POP     HL          ; Zeiger auf Teilbaum des
                            ; zweiten Baums holen
        JP      Z,FERTIG    ; Zeiger stimmen ueberein, also
                            ; stimmen die Teilbaeume ueberein
        LD      D,B         ; Zeiger auf den Teilbaum des
        LD      E,C         ; ersten Baums kopieren
        CALL    VERGL       ; Teilbaeume rekursiv vergleichen
FERTIG: POP     DE          ; Zeiger auf zweiten Baum holen
        POP     HL          ; Zeiger auf ersten Baum holen
        RET
```

Kapitel 23.6

1. Ein Beispiel für die Implementierung dieses Graphen wäre:

```
KNOT0:  DEFB    0           ; Wert des Knotens
        DEFW    KNOT1       ; Kante
        DEFB    'a'         ; Wert der Kante
        DEFW    KNOT4       ; Kante
        DEFW    'f'         ; Wert der Kante
        DEFW    NIL         ; Endemarkierung
KNOT1:  DEFB    1           ; Wert des Knotens
        DEFW    KNOT0       ; Kante
        DEFW    'a'         ; Wert der Kante
        DEFW    KNOT1       ; Kante
        DEFB    'b'         ; Wert der Kante
        DEFW    KNOT2       ; Kante
        DEFW    'c'         ; Wert der Kante
        DEFW    KNOT4       ; Kante
        DEFW    'g'         ; Wert der Kante
        DEFW    NIL         ; Endemarkierung
KNOT2:  DEFB    2           ; Wert des Knotens
        DEFW    KNOT1       ; Kante
        DEFB    'c'         ; Wert der Kante
        DEFW    KNOT3       ; Kante
        DEFW    'd'         ; Wert der Kante
        DEFW    KNOT4       ; Kante
        DEFW    'h'         ; Wert der Kante
```

```
        DEFW    NIL         ; Endemarkierung
KNOT3:  DEFB    3           ; Wert des Knotens
        DEFW    KNOT2       ; Kante
        DEFB    'd'         ; Wert der Kante
        DEFW    KNOT4       ; Kante
        DEFW    'e'         ; Wert der Kante
        DEFW    NIL         ; Endemarkierung
KNOT4:  DEFB    4           ; Wert des Knotens
        DEFW    KNOT0       ; Kante
        DEFW    'f'         ; Wert der Kante
        DEFW    KNOT1       ; Kante
        DEFB    'g'         ; Wert der Kante
        DEFW    KNOT2       ; Kante
        DEFW    'h'         ; Wert der Kante
        DEFW    KNOT3       ; Kante
        DEFW    'e'         ; Wert der Kante
        DEFW    NIL         ; Endemarkierung
```

Kapitel 24.1

1. Der Multiplikand ist »L« Bytes lang, hat also einen Wert kleiner als $256^{»L«}$. Der in Arbeit befindliche Anteil des Multiplikators ist i Bytes lang - wobei i von 1 bis »L« wächst - und hat damit einen Wert kleiner als 256^{i}. Das Teilprodukt besitzt damit einen Wert kleiner als $256^{»L«+i}$, ist also durch »L«+i Bytes darstellbar. Beim Linksverschieben des Zwischenergebnisses reicht es deshalb aus, »L«+i Bytes zu verschieben. Der beim Addieren eventuell anfallende Übertrag läuft höchstens über i Bytes (statt über »L« Bytes) weiter.

   Die aktuelle Länge i des Multiplikators besetzen wir jeweils im Hauptprogramm; wir wählen dazu das C'-Register. Die modifizierten Routinen lauten:

```
ADD:    PUSH    BC          ; Register-
        PUSH    DE          ; inhalte
        PUSH    HL          ; sichern
        OR      A           ; Uebertrag-Flag loeschen
        LD      B,LAENGE    ; Laenge des Multiplikanden laden
ADDB:   LD      A,(DE)      ; Byte des Multiplikanden holen
        ADC     A,(HL)      ; Byte des Zwischenergebnisses
                            ; hinzuaddieren
        LD      (HL),A      ; Byte ins Zwischenergebnis
                            ; zurueckschreiben
        INC     DE          ; auf naechstes Byte des
                            ; Multiplikanden zeigen
        INC     HL          ; auf naechstes Byte des
```

```
                                    ; Zwischenergebnisses zeigen
        DJNZ    ADDB                ; alle Bytes des Multiplikanden
                                    ; zum Zwischenergebnis addieren
        LD      B,C                 ; Optimierung!
                                    ; restliche Laenge des
                                    ; Zwischenergebnisses laden
        JP      NC,FERTIG           ; kein Uebertrag mehr zu
                                    ; beruecksichtigen
UEBERT: INC     (HL)                ; Uebertrag von vorhergehender
                                    ; Stelle beruecksichtigen
        JP      NZ,FERTIG           ; kein weiterer Uebertrag
                                    ; zu beruecksichtigen
        INC     HL                  ; auf naechstes Byte des
                                    ; Zwischenergebnisses zeigen
        DJNZ    UEBERT              ; Uebertrag eventuell durch alle
                                    ; Bytes des Zwischenergebnisses
                                    ; ziehen
FERTIG: POP     HL                  ; alle
        POP     DE                  ; Register
        POP     BC                  ; restaurieren
        RET
SCHIEB: PUSH    HL                  ; Registerinhalte
        PUSH    BC                  ; sichern
        LD      A,LAENGE            ; Laenge des Multiplikanden
        ADD     A,C                 ; aktuelle Laenge des
                                    ; Multiplikators addieren
        LD      B,A                 ; Laenge des Zwischenergebnisses
                                    ; laden
        OR      A                   ; Uebertrag-Flag loeschen
SBYTE:  RL      (HL)                ; Byte um ein Bit
                                    ; linksverschieben, alten Wert
                                    ; des Uebertrag-Flags einfuegen,
                                    ; hoechstes Bit ins
                                    ; Uebertrag-Flag bringen
        INC     HL                  ; auf naechstes Byte des
                                    ; Zwischenergebnisses zeigen
        DJNZ    SBYTE               ; alle Bytes des
                                    ; Zwischenergebnisses bearbeiten
        POP     BC                  ; Register
        POP     HL                  ; wiederherstellen
        RET
MULT:   PUSH    DE                  ; Zeiger auf Multiplikand und
        PUSH    BC                  ; Zeiger auf Ergebnis auf den
```

```
                                    ; Stapel bringen zwecks
                                    ; Einbringung in sekundaeren
                                    ; Registersatz
        LD      DE,LAENGE           ; Zeiger auf Byte vor dem
        ADD     HL,DE               ; hoechstwertigen Byte des
                                    ; Multiplikators berechnen
        EXX                         ; sekundaeren Registersatz holen
        POP     HL                  ; Zeiger auf Ergebnis und
        POP     DE                  ; Zeiger auf Multiplikand holen
        CALL    LOESCH              ; Akkumulator loeschen
        LD      C,0                 ; aktuelle Laenge des
                                    ; Multiplikators ruecksetzen
        EXX                         ; primaeren Registersatz holen
        LD      B,LAENGE            ; Anzahl der Bytes des
                                    ; Multiplikators laden
MULT:   DEC     HL                  ; auf naechstes Byte des
                                    ; Multiplikators zeigen
        EXX                         ; sekundaeren Registersatz holen
        LD      B,8                 ; Anzahl der Bits pro Byte laden
        INC     C                   ; aktuelle Laenge des
                                    ; Multiplikators erhoehen
BMULT:  CALL    SCHIEB              ; Zwischenergebnis um ein Bit
                                    ; linksschieben
        EXX                         ; primaeren Registersatz holen
        RLC     (HL)                ; Bit des Multiplikators holen
        EXX                         ; sekundaeren Registersatz holen
        CALL    C,ADD               ; Bit war gesetzt, Multiplikand
                                    ; zum Zwischenergebnis addieren
        DJNZ    BMULT               ; alle Bits dieses Bytes des
                                    ; Multiplikators verarbeiten
        EXX                         ; primaeren Registersatz holen
        DJNZ    MULT                ; alle Bytes des Multiplikators
                                    ; verarbeiten
```

Im optimierten Verfahren von Booth ist in analoger Weise zu verfahren.

Kapitel 24.4

1. Wir invertieren das Vorzeichen, das sich im höherwertigen Nibble von Byte 9 (relativ ab 0 gerechnet) befindet:

```
        LD      DE,9                ; auf Vorzeichen
        ADD     HL,DE               ; zeigen
```

```
        LD      A,10000000B ; Maske zum Invertieren von Bit 7
        XOR     (HL)        ; Vorzeichen umkehren
        LD      (HL),A      ; Vorzeichen zurueckschreiben
```

2. Für die Multiplikation der Beträge ist die Größe LAENGE als 9 zu vereinbaren. Wir führen zunächst die Multiplikation der Beträge durch und prüfen anschließend, ob das Ergebnis wieder in 9 Bytes Platz findet; andernfalls springen wir eine Überlaufadresse an. Anschließend berechnen wir aus den Vorzeichen der Operanden das Vorzeichen des Produkts; hierzu werden die Vorzeichenbits mit XOR verknüpft, denn zwei gleiche Vorzeichen resultieren in positivem Vorzeichen (codiert durch 0), ungleiche in negativem Vorzeichen (codiert durch 1).

```
LAENGE  EQU     9           ; Laenge der Betraege
        PUSH    HL          ; Register-
        PUSH    DE          ; inhalte
        PUSH    BC          ; sichern
        CALL    MULT        ; Betraege multiplizieren
        POP     HL          ; Zeiger auf Ergebnis holen
        LD      DE,18       ; Zeiger vor
        ADD     HL,DE       ; Betrag daraus berechnen
        LD      B,9         ; Anzahl der zu pruefenden Bytes
        XOR     A           ; Akkumulator loeschen
TEST:   DEC     HL          ; auf naechstes Byte zeigen
        OR      (HL)        ; Byte auf Null testen
        JP      NZ,UEBERL   ; nicht Null, Ueberlauf!
        DJNZ    TEST        ; alle 9 fuehrenden Bytes testen
        EX      DE,HL       ; Zeiger auf Vorzeichen des
                            ; Ergebnisses sichern
        POP     HL          ; Zeiger auf Multiplikand holen
        LD      BC,9        ; auf Vorzeichen des
        ADD     HL,BC       ; Multiplikanden zeigen
        LD      A,(HL)      ; Vorzeichen des Multiplikanden
                            ; holen
        POP     HL          ; Zeiger auf Multiplikator holen
        ADD     HL,BC       ; auf Vorzeichen des
                            ; Multiplikators zeigen
        XOR     (HL)        ; Vorzeichen des Multiplikators
                            ; mit Vorzeichen des
                            ; Multiplikanden verknuepfen
        LD      (DE),A      ; Vorzeichen des Ergebnisses
                            ; eintragen
```

Kapitel 26.2

1. Wir nehmen wieder an, daß der Spaltenindex j, der ab 0 gezählt wird, im A-Register steht. Die Adressen je zweier direkt aufeinanderfolgender Zeichen in derselben Spalte unterscheiden sich um den Wert 64, das ist die Länge einer Zeile. Das Programm lautet damit:

```
        LD      HL,3C00H    ; Anfangsadresse des
                            ; Bildschirmspeichers
        LD      D,0         ; Spaltenindex zu
        LD      E,A         ; Relativadresse machen
        ADD     HL,DE       ; Adresse des obersten Zeichens
                            ; in Spalte j berechnen
        LD      DE,64       ; Anzahl der Zeichen pro Zeile
        LD      B,16        ; Anzahl der Zeilen
LOESCH: LD      (HL),' '    ; Zeichen loeschen
        ADD     HL,DE       ; eine Zeile tiefergehen
        DJNZ    LOESCH      ; gesamte Spalte loeschen
```

2. Das auszugebende Zeichen stehe wieder im D-Register. Im modifizierten Programm wird zunächst auf Fehler und dann auf Warten getestet:

```
        LD      E,01100000B ; Maske fuer Fehlerbits
        LD      HL,37E8H    ; Geraeteadresse
WARTE:  LD      A,(HL)      ; Status holen
        LD      B,A         ; und sichern
        AND     E           ; auf Fehler testen
        JP      NZ,FEHLER   ; Fehler aufgetreten
        BIT     7,B         ; auf Warten testen
        JP      NZ,WARTE    ; warten
        LD      (HL),D      ; Zeichen ausgeben
```

Kapitel 26.3

1. Wir nehmen die Adresse des Datenblocks im HL-Register an, seine Länge im DE-Register. Das Ausgeben eines Datenblocks erfolgt dann durch folgendes Programm:

```
        LD      B,11100000B ; Maske fuer Status
        LD      C,62H       ; Portadresse des Druckers
        INC     DE          ; Zaehler korrigieren
        JP      TEST        ; Schleife abweisend machen
WARTE:  IN      A,(C)       ; Status lesen
        AND     B           ; Status pruefen
```

```
        JP      NZ,WARTE    ; Drucker nicht bereit
        LD      A,(HL)      ; auszugebendes Zeichen holen
        OUT     (C),A       ; und ausgeben
        INC     HL          ; auf naechstes Zeichen zeigen
TEST:   DEC     DE          ; Zaehler vermindern
        LD      A,D         ; und auf Null
        OR      E           ; testen
        JP      NZ,WARTE    ; Rest des Datenblocks ausgeben
```

2. Wir testen zunächst auf echte Fehler, dann auf Warten:

```
        LD      E,01100000B ; Maske fuer Fehler
        LD      C,62H       ; Portadresse des Druckers
        LD      HL,95B0H    ; Anfangsadresse des Datenblocks
        LD      B,80        ; Laenge des Datenblocks
WARTE:  IN      A,(C)       ; Status lesen
        LD      D,A         ; und sichern
        AND     E           ; auf Fehler testen
        JP      NZ,FEHLER   ; Fehler
        BIT     7,D         ; auf Warten testen
        JP      NZ,WARTE    ; Warten
        OUTI                ; Zeichen ausgeben, Zeiger und
                            ; Zaehler korrigieren
        JP      NZ,WARTE    ; Rest des Datenblocks ausgeben
```

Kapitel 27.7

1. Wird das Unterprogramm durch den Produzenten unterbrochen und ist der Puffer gerade voll, so erhält der Produzent die eigentlich unzutreffende Fehlermeldung, daß der Puffer voll ist (es wird ja durch das Unterprogramm gerade ein Platz im Puffer freigemacht).

   Wir nehmen zunächst an, daß der Produzent alle Register einschließlich der Flags intakt läßt.

   Würde zwischen CALL LEER und JP Z,LENDE unterbrochen, so würde fälschlicherweise wegen leeren Puffers abgebrochen, obwohl der Puffer dann gar nicht mehr leer wäre. Diese beiden Befehle bilden also einen kritischen Bereich.

   Zwischen JP Z,LENDE und LD HL,(KZEIG) finden keine Operationen statt, die voneinander abhängen. Die PUSH-Befehle gehören deshalb nicht zum kritischen Bereich.

   Der Konsumentenzeiger wird durch den Produzenten keinesfalls verändert. Somit gehören auch die Befehle LD HL,(KZEIG) bis LD (KZEIG),HL nicht zum kritischen Bereich.

   Nun folgt die Phase, in der bei leerem Puffer die Puffer-Zeiger rückgesetzt werden. Dies darf nicht unterbrochen werden, da sonst die Voraussetzung (leerer Puffer) verletzt wird. Der Bereich von LD DE,(PZEIG) bis CALL Z,INIT ist damit kritisch.

Die POP-Befehle sind dagegen wieder unkritisch, da der Produzent alle Register wiederherstellt.

Das gesamte Unterprogramm enthält damit zwei kritische Bereiche (ein bedingter Sprung kann in einen bedingten Unterprogramm-Rücksprung verwandelt werden):

```
LEERE:  DI                      ; maskierbare Unterbrechungen
                                ; sperren
        CALL    LEER            ; pruefen, ob Puffer leer
        EI                      ; maskierbare Unterbrechungen
                                ; zulassen
        RET     Z               ; Puffer leer, Fehler
        PUSH    HL              ; Registerinhalte
        PUSH    DE              ; sichern
        LD      HL,(KZEIG)      ; Konsumenten-Zeiger holen
        LD      A,(HL)          ; Zeichen aus Puffer nehmen
        PUSH    AF              ; Zeichen und Null-Flag sichern
        INC     HL              ; auf naechstes Zeichen zeigen
        LD      (KZEIG),HL      ; Konsumenten-Zeiger sichern
        DI                      ; maskierbare Unterbrechungen
                                ; sperren
        LD      DE,(PZEIG)      ; Produzenten-Zeiger holen
        OR      A               ; pruefen, ob Puffer-Zeiger
        SBC     HL,DE           ; uebereinstimmen
        CALL    Z,INIT          ; Puffer leer,
                                ; Puffer-Zeiger ruecksetzen
        EI                      ; maskierbare Unterbrechungen
                                ; zulassen
        POP     AF              ; alle
        POP     DE              ; Register
        POP     HL              ; restaurieren
        EI                      ; maskierbare Unterbrechungen
                                ; zulassen
        RET
```

Ganz anders liegt der Fall, wenn wir nicht wissen, welche Register der Produzent zerstört. In diesem Fall darf zwischen dem Eintritt ins Unterprogramm LEERE und dem Abschluß der Sicherungsoperationen (PUSH) keine Unterbrechung erfolgen. Ebenso darf nach dem ersten Restaurierungsbefehl (POP) bis zum Programmaustritt keine Unterbrechung erfolgen. Dies bedeutet, daß Unterbrechungen vor dem Eintritt außerhalb des Unterprogramms gesperrt werden müssen und erst wieder nach Austritt außerhalb des Unterprogramms zugelassen werden.

Zwischen den Befehlen PUSH DE und LD HL,(KZEIG) darf unterbrochen werden.

Direkte Abhängigkeiten bestehen zwischen allen Befehlen von LD HL,(KZEIG) bis CALL Z,INIT. Dieser Abschnitt ist damit kritisch.

Zwischen CALL Z,INIT und POP AF darf dagegen wieder unterbrochen werden.

Das Unterprogramm lautet also für diesen Fall:

```
LEERE:  DI                      ; maskierbare Unterbrechungen
                                ; sperren
        CALL    LEER            ; pruefen, ob Puffer leer
        RET     Z               ; Puffer leer, Fehler
        PUSH    HL              ; Registerinhalte sichern
        EI                      ; maskierbare Unterbrechungen
                                ; zulassen
        PUSH    DE              ; Registerinhalt sichern
        DI                      ; maskierbare Unterbrechungen
                                ; sperren
        LD      HL,(KZEIG)      ; Konsumenten-Zeiger holen
        LD      A,(HL)          ; Zeichen aus Puffer nehmen
        PUSH    AF              ; und sichern
        INC     HL              ; auf naechstes Zeichen zeigen
        LD      (KZEIG),HL      ; Konsumenten-Zeiger sichern
        LD      DE,(PZEIG)      ; Produzenten-Zeiger holen
        OR      A               ; pruefen, ob Puffer-Zeiger
        SBC     HL,DE           ; uebereinstimmen
        CALL    Z,INIT          ; Puffer leer,
                                ; Puffer-Zeiger ruecksetzen
        EI                      ; maskierbare Unterbrechungen
        NOP                     ; zulassen
        DI                      ; maskierbare Unterbrechungen
                                ; sperren
        POP     AF              ; alle
        POP     DE              ; Register
        POP     HL              ; restaurieren
        RET
```

2. Echte eintrittsinvariante Unterprogramme arbeiten nur auf Daten, auf die sie externen Zugriff haben. Die Gültigkeit von Daten in einem eintrittsinvarianten Unterprogramm wird durch Unterbrechungen deshalb nicht beeinträchtigt. Prinzipiell kann ein eintrittsinvariantes Unterprogramm deshalb an jeder beliebigen Stelle unterbrochen werden.

   Es kann dabei allerdings zu logischen Fehlern kommen, zum Beispiel wenn zwischen einem Test auf Gültigkeit einer Bedingung und einer Aktion, die diese Bedingung voraussetzt, eine Unterbrechung erfolgt, welche die Bedingung beeinflußt.

   Beim Z80 verfügen wir nicht über die Möglichkeit, echte eintrittsinvariante Unterprogramme zu schreiben. Kritisch sind deshalb stets solche Bereiche, in denen Daten gesichert

(PUSH) oder restauriert (POP) werden beziehungsweise solche, in denen voneinander abhängige Befehle vorkommen (LD HL,2337H ... LD (HL),A).

Kapitel 28.1

1. Wenn wir ein Programm ohne absolute Sprünge und ohne Unterprogrammaufrufe aufbauen wollen, so können wir vom Sprungpunkt aus jeweils nur um 128 Bytes zurück oder um 127 Bytes vorwärts springen. In manchen Fällen können die einzelnen Programmteile aber gar nicht so angeordnet werden, daß alle Sprungziele diesen Bedingungen genügen. Eine Abhilfe stellt das Anspringen von Hilfs-Befehlen dar, die selbst wieder Sprünge sind, zum Beispiel

```
        JR      HILF        ; Anspringen des Hilfs-Befehls
        :
        :
        :
HILF:   JR      ZIEL        ; Ansprung des eigentlichen Ziels
        :
        :
        :
ZIEL:                       ; eigentliches Sprungziel
```

Wir werden in den Unterkapiteln 28.2 und 28.3 eine Methode erarbeiten, die den Einsatz indirekter (absoluter) Sprünge in verschiebbaren Programmen erlaubt.

Kapitel 28.2

1. Die entsprechenden Routinen lauten:

```
BBASIS: POP     BC      ; Basis-Adresse ins BC-Register
                        ; bringen
        PUSH    BC      ; und wieder als
                        ; Rueckkehradresse eintragen
        RET             ; Ruecksprung zur Basis-Adresse
DBASIS: POP     DE      ; Basis-Adresse ins DE-Register
                        ; bringen
        PUSH    DE      ; und wieder als
                        ; Rueckkehradresse eintragen
        RET             ; Ruecksprung zur Basis-Adresse
```

Kapitel 28.3

1. Statt eines relativen Sprungs wird folgendes Programmstück ins Programm eingefügt:

```
        INC     DE          ; Relativadresse um
        INC     DE          ; zwei erhoehen
        CALL    HBASIS      ; Basis-Adresse beschaffen
PBASIS: ADD     HL,DE       ; Sprungadresse berechnen
        JP      (HL)        ; Ziel anspringen
BASIS:
```

Da HBASIS die Adresse PBASIS des folgenden Befehls liefert, mußte noch die Länge des Objekt-Codes zwischen dieser Pseudo-Basis und der auf den indirekten Sprung folgenden echten Basis BASIS zur Relativadresse addiert werden.

2. Das verschiebbare Programmstück lautet:

```
        CALL    HBASIS          ; Basis-Adresse beschaffen
BASIS:  LD      DE,RUECK-BASIS      ; Relativadresse zwischen
                                    ; Basis-Adresse und
                                    ; Rueckkehradresse
        ADD     HL,DE           ; Rueckkehradresse berechnen
        PUSH    HL              ; Rueckkehradresse ablegen
        LD      DE,UP-RUECK     ; Relativadresse zwischen
                                ; Rueckkehradresse und
                                ; Unterprogramm-Aufrufadresse
        ADD     HL,DE           ; Unterprogramm-Aufrufadresse
                                ; berechnen
        JP      (HL)            ; Unterprogramm aufrufen
RUECK:                          ; Rueckkehradresse
        :
        :
        :
UP:                             ; Unterprogramm
        :
        :
        :
        RET                     ; Rueckkehr zur Adresse RUECK
```

Anhang A

Verzeichnis der Assembler-Befehle (nach Funktionsgruppen geordnet)

8-Bit-Lade-Befehle

LD	A,	A B C D E H L xx	(xxyy) (BC) (DE)	(HL)	(IX+zz)	(IY+zz)	I R
LD	B,	A B C D E H L xx		(HL)	(IX+zz)	(IY+zz)	
LD	C,	A B C D E H L xx		(HL)	(IX+zz)	(IY+zz)	
LD	D,	A B C D E H L xx		(HL)	(IX+zz)	(IY+zz)	
LD	E,	A B C D E H L xx		(HL)	(IX+zz)	(IY+zz)	
LD	H,	A B C D E H L xx		(HL)	(IX+zz)	(IY+zz)	
LD	L,	A B C D E H L xx		(HL)	(IX+zz)	(IY+zz)	
LD	(xxyy),	A					
LD	(BC),	A					
LD	(DE),	A					
LD	(HL),	A B C D E H L xx					
LD	(IX+zz),	A B C D E H L xx					
LD	(IY+zz),	A B C D E H L xx					
LD	I,	A					
LD	R,	A					

Die Flags werden durch diese Befehle nicht verändert, außer bei:

LD A,I

S: gesetzt, falls Inhalt des I-Registers negativ ist
Z: gesetzt, falls Inhalt des I-Registers Null ist
H: rückgesetzt
P: enthält den Inhalt von IFF2 (interrupt-flip-flop 2)

N: rückgesetzt
C: unverändert

```
LD    A,R
```

S: gesetzt, falls Inhalt des R-Registers negativ ist
Z: gesetzt, falls Inhalt des R-Registers Null ist
H: rückgesetzt
P: enthält den Inhalt von IFF2 (interrupt-flip-flop 2)
N: rückgesetzt
C: unverändert

16-Bit-Lade-Befehle

```
LD    BC,                                    xxyy (xxyy)
LD    DE,                                    xxyy (xxyy)
LD    HL,                                    xxyy (xxyy)
LD    SP,                     HL    IX IY    xxyy (xxyy)
LD    IX,                                    xxyy (xxyy)
LD    IY,                                    xxyy (xxyy)
LD    (xxyy),       BC DE HL SP IX IY
PUSH            AF BC DE HL    IX IY
POP             AF BC DE HL    IX IY
```

Die Flags werden durch diese Befehle nicht verändert.

Austausch-Befehle

```
EX    (SP),HL
EX    (SP),IX
EX    (SP),IY
EX    AF,AF'
EX    DE,HL
EXX
```

Dic Flags werden durch diese Befehle nicht verändert.

Befehle für blockweises Bewegen

```
LDI
LDD
```

Die Flags erhalten folgende Werte:

S: unverändert
Z: unverändert
H: rückgesetzt
P: rückgesetzt, falls Inhalt des BC-Registers Null wurde
N: rückgesetzt
C: unverändert

LDIR
LDDR

Die Flags erhalten folgende Werte:

S: unverändert
Z: unverändert
H: rückgesetzt
P: rückgesetzt
N: rückgesetzt
C: unverändert

Such-Befehle

CPI
CPD
CPIR
CPDR

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: rückgesetzt, falls Borgen von Bit 4 nötig war
P: rückgesetzt, falls Inhalt des BC-Registers Null wurde
N: gesetzt
C: unverändert

8-Bit-Arithmetik- und Logik-Befehle

ADD A, A B C D E H L xx (HL) (IX+zz) (IY+zz)
ADC A, A B C D E H L xx (HL) (IX+zz) (IY+zz)

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: gesetzt, falls Übertrag von Bit 3
P: gesetzt, falls Überlauf auftrat
N: rückgesetzt
C: gesetzt, falls Übertrag von Bit 7

```
SUB       A B C D E H L xx (HL) (IX+zz) (IY+zz)
SBC  A,   A B C D E H L xx (HL) (IX+zz) (IY+zz)
CP        A B C D E H L xx (HL) (IX+zz) (IY+zz)
```

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: rückgesetzt, falls Borgen von Bit 4 nötig war
P: gesetzt, falls Überlauf auftrat
N: gesetzt
C: gesetzt, falls Borgen nötig war

```
AND       A B C D E H L xx (HL) (IX+zz) (IY+zz)
OR        A B C D E H L xx (HL) (IX+zz) (IY+zz)
XOR       A B C D E H L xx (HL) (IX+zz) (IY+zz)
```

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: gesetzt
P: gesetzt, falls Parität gerade
N: rückgesetzt
C: rückgesetzt

```
INC       A B C D E H L (HL) (IX+zz) (IY+zz)
```

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: gesetzt, falls Übertrag von Bit 3
P: gesetzt, falls Überlauf auftrat

N: rückgesetzt
C: unverändert

DEC A B C D E H L (HL) (IX+zz) (IY+zz)

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: rückgesetzt, falls Borgen von Bit 4 nötig war
P: gesetzt, falls Überlauf auftrat
N: gesetzt
C: unverändert

SCF

Die Flags erhalten folgende Werte:

S: unverändert
Z: unverändert
H: rückgesetzt
P: unverändert
N: rückgesetzt
C: gesetzt

CCF

Die Flags erhalten folgende Werte:

S: unverändert
Z: unverändert
H: alter Wert des Übertrag-Flags
P: unverändert
N: rückgesetzt
C: gesetzt, falls Übertrag-Flag vorher nicht gesetzt

CPL

Die Flags erhalten folgende Werte:

S: unverändert
Z: unverändert
H: gesetzt

P: unverändert
N: gesetzt
C: unverändert

NEG

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: rückgesetzt, falls Borgen von Bit 4 nötig war
P: gesetzt, falls Überlauf auftrat
N: gesetzt
C: gesetzt, falls Borgen nötig war

DAA

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: gesetzt, falls Übertrag von Bit 3 und N-Flag rückgesetzt oder falls Borgen von Bit 4 nötig war und N-Flag gesetzt
P: gesetzt, falls Parität gerade
N: unverändert
C: siehe nachfolgende Tabelle

N-Flag (vorher)	0	0	0	0	0	0	0	0	0	1	1	1	1
C-Flag (vorher)	0	0	0	0	0	0	1	1	1	0	0	1	1
H-Flag (vorher)	0	0	1	0	0	1	0	0	1	0	1	0	1
Wert in Bit 7-4 des A-Registers (vorher)	0-9	0-8	0-9	A-F	9-F	A-F	0-2	0-2	0-3	0-9	0-8	7-F	6-F
Wert in Bit 3-0 des A-Registers (vorher)	0-9	A-F	0-3	0-9	A-F	0-3	0-9	A-F	0-3	0-9	6-F	0-9	6-F
C-Flag (nachher)	0	0	0	1	1	1	1	1	1	0	0	1	1
zum Inhalt des A-Registers addierter Wert	00H	06H	06H	60H	66H	66H	60H	66H	66H	00H	FAH	A0H	9AH

16-Bit-Arithmetik-Befehle

```
ADD   HL,      BC DE HL SP
ADD   IX,      BC DE    SP IX
ADD   IY,      BC DE    SP    IY
```

Die Flags erhalten folgende Werte:

S: unverändert
Z: unverändert
H: gesetzt, falls Übertrag von Bit 11
P: unverändert
N: rückgesetzt
C: gesetzt, falls Übertrag von Bit 15

```
ADC   HL,      BC DE HL SP
```

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: gesetzt, falls Übertrag von Bit 11
P: gesetzt, falls Überlauf auftrat
N: rückgesetzt
C: gesetzt, falls Übertrag von Bit 15

```
SBC   HL,      BC DE HL SP
```

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: rückgesetzt, falls Borgen von Bit 12 nötig war
P: gesetzt, falls Überlauf auftrat
N: gesetzt
C: gesetzt, falls Borgen nötig war

```
INC            BC DE HL SP IX IY
DEC            BC DE HL SP IX IY
```

Die Flags werden durch diese Befehle nicht verändert.

Rotations- und Verschiebe-Befehle

RLC A B C D E H L (HL) (IX+zz) (IY+zz)
RL A B C D E H L (HL) (IX+zz) (IY+zz)
SLA A B C D E H L (HL) (IX+zz) (IY+zz)

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: rückgesetzt
P: gesetzt, falls Parität gerade
N: rückgesetzt
C: Bit 7 des ursprünglichen Inhalts

RRC A B C D E H L (HL) (IX+zz) (IY+zz)
RR A B C D E H L (HL) (IX+zz) (IY+zz)
SRA A B C D E H L (HL) (IX+zz) (IY+zz)
SRL A B C D E H L (HL) (IX+zz) (IY+zz)

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: rückgesetzt
P: gesetzt, falls Parität gerade
N: rückgesetzt
C: Bit 0 des ursprünglichen Inhalts

RLCA
RLA

Die Flags erhalten folgende Werte:

S: unverändert
Z: unverändert
H: rückgesetzt
P: unverändert
N: rückgesetzt
C: Bit 7 des ursprünglichen Inhalts

RRCA
RRA

Die Flags erhalten folgende Werte:

S: unverändert
Z: unverändert
H: rückgesetzt
P: unverändert
N: rückgesetzt
C: Bit 0 des ursprünglichen Inhalts

RLD
RRD

Die Flags erhalten folgende Werte:

S: gesetzt, falls Inhalt des A-Registers negativ wurde
Z: gesetzt, falls Inhalt des A-Registers Null wurde
H: rückgesetzt
P: gesetzt, falls Parität gerade
N: rückgesetzt
C: unverändert

Bit-Manipulations-Befehle

```
BIT  0,   A B C D E H L (HL) (IX+zz) (IY+zz)
BIT  1,   A B C D E H L (HL) (IX+zz) (IY+zz)
BIT  2,   A B C D E H L (HL) (IX+zz) (IY+zz)
BIT  3,   A B C D E H L (HL) (IX+zz) (IY+zz)
BIT  4,   A B C D E H L (HL) (IX+zz) (IY+zz)
BIT  5,   A B C D E H L (HL) (IX+zz) (IY+zz)
BIT  6,   A B C D E H L (HL) (IX+zz) (IY+zz)
BIT  7,   A B C D E H L (HL) (IX+zz) (IY+zz)
```

Die Flags erhalten folgende Werte:

S: unbekannt
Z: gesetzt, falls das entsprechende Bit Null ist
H: gesetzt
P: unbekannt
N: rückgesetzt
C: unverändert

```
SET   0,      A B C D E H L (HL) (IX+zz) (IY+zz)
SET   1,      A B C D E H L (HL) (IX+zz) (IY+zz)
SET   2,      A B C D E H L (HL) (IX+zz) (IY+zz)
SET   3,      A B C D E H L (HL) (IX+zz) (IY+zz)
SET   4,      A B C D E H L (HL) (IX+zz) (IY+zz)
SET   5,      A B C D E H L (HL) (IX+zz) (IY+zz)
SET   6,      A B C D E H L (HL) (IX+zz) (IY+zz)
SET   7,      A B C D E H L (HL) (IX+zz) (IY+zz)
RES   0,      A B C D E H L (HL) (IX+zz) (IY+zz)
RES   1,      A B C D E H L (HL) (IX+zz) (IY+zz)
RES   2,      A B C D E H L (HL) (IX+zz) (IY+zz)
RES   3,      A B C D E H L (HL) (IX+zz) (IY+zz)
RES   4,      A B C D E H L (HL) (IX+zz) (IY+zz)
RES   5,      A B C D E H L (HL) (IX+zz) (IY+zz)
RES   6,      A B C D E H L (HL) (IX+zz) (IY+zz)
RES   7,      A B C D E H L (HL) (IX+zz) (IY+zz)
```

Die Flags werden durch diese Befehle nicht verändert.

Sprung-Befehle

```
JP            xxyy (HL) (IX) (IY)
JP    NZ,     xxyy
JP    Z,      xxyy
JP    NC,     xxyy
JP    C,      xxyy
JP    PO,     xxyy
JP    PE,     xxyy
JP    P,      xxyy
JP    M,      xxyy
JR    tt
JR    NZ,     tt
JR    Z,      tt
JR    NC,     tt
JR    C,      tt
DJNZ          tt
```

Die Flags werden durch diese Befehle nicht verändert.

Unterprogramm-Befehle

```
CALL          xxyy
CALL  NZ,     xxyy
CALL  Z,      xxyy
CALL  NC,     xxyy
CALL  C,      xxyy
CALL  PO,     xxyy
CALL  PE,     xxyy
CALL  P,      xxyy
CALL  M,      xxyy
RET
RET   NZ
RET   Z
RET   NC
RET   C
RET   PO
RET   PE
RET   P
RET   M
RETI
RETN
RST           00H 08H 10H 18H 20H 28H 30H 38H
```

Die Flags werden durch diese Befehle nicht verändert.

Kontroll-Befehle

```
NOP
HALT
DI
EI
IM 0
IM 1
IM 2
```

Die Flags werden durch diese Befehle nicht verändert.

Ein-/Ausgabe-Befehle

```
IN    A,(C)
IN    B,(C)
```

```
IN      C,(C)
IN      D,(C)
IN      E,(C)
IN      H,(C)
IN      L,(C)
```

Die Flags erhalten folgende Werte:

S: gesetzt, falls Resultat negativ ist
Z: gesetzt, falls Resultat Null ist
H: rückgesetzt
P: gesetzt, falls Parität gerade
N: rückgesetzt
C: unverändert

```
IN      A,(xx)
OUT     (xx),A
OUT     (C),A
OUT     (C),B
OUT     (C),C
OUT     (C),D
OUT     (C),E
OUT     (C),H
OUT     (C),L
```

Die Flags werden durch diese Befehle nicht verändert.

```
INI
IND
OUTI
OUTD
```

Die Flags erhalten folgende Werte:

S: unbekannt
Z: gesetzt, falls Inhalt des B-Registers Null wurde
H: unbekannt
P: unbekannt
N: gesetzt
C: unverändert

```
INIR
INDR
```

OTIR
OTDR

Die Flags erhalten folgende Werte:

S: unbekannt
Z: gesetzt
H: unbekannt
P: unbekannt
N: gesetzt
C: unverändert

Anhang B

Verzeichnis der Assembler-Befehle (lexikalisch sortiert)

Das nachfolgende Verzeichnis enthält für jeden Assembler-Befehl in Standard ZILOG Z80 Notation einen Eintrag. Die Einträge sind nach dem Befehlsnamen lexikalisch sortiert. Einträge mit gleichem Befehlsnamen sind nach dem ersten Operanden lexikalisch sortiert, Einträge mit gleichem Befehlsnamen und gleichem ersten Operanden nach dem zweiten Operanden. Die lexikalische Ordnung folgt dem ASCII-Code.

Direkte Operanden vom Typ »Byte« und direkte Portadressen werden durch xx bezeichnet, direkte Operanden vom Typ »Wort« und direkte Speicheradressen durch xxyy. Relativadressen bezüglich eines Indexregisters werden durch zz bezeichnet. Relativadressen in einem JR- oder DJNZ-Befehl werden durch tt bezeichnet.

Jeder Eintrag des Verzeichnisses enthält (von links nach rechts) folgende Komponenten:

- Befehlsname
- Operanden (falls welche vorhanden)
- Objekt-Code (1 bis 4 Bytes in Hex-Darstellung; der Übersichtlichkeit wegen werden führende Nullen und abschließendes »H« unterdrückt)
- Die Ausführungszeit in Takt-Zyklen (es kommt vor, daß mehrere Ausführungszeiten genannt werden, wobei die tatsächliche Ausführungszeit vom Ablauf der jeweiligen Operation abhängt; Beispiel: relative Sprünge)
- Eine Beschreibung der ausgelösten Operation in formaler Notation

In der Beschreibungssprache werden folgende Symbole verwendet:

<—	Zuweisung
< >	Inhalt eines Registers, eines Ports oder einer Speicherzelle
( )	durch Speicheradresse bezeichnete Speicherzelle
[ ]	durch Portadresse bezeichneter Port
&	Konkatenation von Registern oder Speicherzellen bzw. -inhalten
and	bitweise Konjunktion

or	bitweise Disjunktion
xor	bitweise exklusive Disjunktion
not	bitweise Negation

Durch Indizes werden die einzelnen Bits eines Registers oder einer Speicherzelle bzw. eines Register- oder Speicherinhalts bezeichnet. Die Indizes »H« beziehungsweise »L« stehen für den höherwertigen beziehungsweise niederwertigen Anteil einer Größe vom Typ »Wort«.

ADC	A,(HL)	8E	7	A<-<A>+<(<HL>)>+<CY>
ADC	A,(IX+zz)	DD 8E zz	19	A<-<A>+<(<IX>+zz)>+<CY>
ADC	A,(IY+zz)	FD 8E zz	19	A<-<A>+<(<IY>+zz)>+<CY>
ADC	A,A	8F	4	A<-<A>+<A>+<CY>
ADC	A,B	88	4	A<-<A>+<B>+<CY>
ADC	A,C	89	4	A<-<A>+<C>+<CY>
ADC	A,D	8A	4	A<-<A>+<D>+<CY>
ADC	A,E	8B	4	A<-<A>+<E>+<CY>
ADC	A,H	8C	4	A<-<A>+<H>+<CY>
ADC	A,L	8D	4	A<-<A>+<L>+<CY>
ADC	A,xx	CE xx	7	A<-<A>+xx+<CY>
ADC	HL,BC	ED 4A	15	HL<-<HL>+<BC>+<CY>
ADC	HL,DE	ED 5A	15	HL<-<HL>+<DE>+<CY>
ADC	HL,HL	ED 6A	15	HL<-<HL>+<HL>+<CY>
ADC	HL,SP	ED 7A	15	HL<-<HL>+<SP>+<CY>
ADD	A,(HL)	86	7	A<-<A>+<(<HL>)>
ADD	A,(IX+zz)	DD 86 zz	19	A<-<A>+<(<IX>+zz)>
ADD	A,(IY+zz)	FD 86 zz	19	A<-<A>+<(<IY>+zz)>
ADD	A,A	87	4	A<-<A>+<A>
ADD	A,B	80	4	A<-<A>+<B>
ADD	A,C	81	4	A<-<A>+<C>
ADD	A,D	82	4	A<-<A>+<D>
ADD	A,E	83	4	A<-<A>+<E>
ADD	A,H	84	4	A<-<A>+<H>
ADD	A,L	85	4	A<-<A>+<L>
ADD	A,xx	C6 xx	7	A<-<A>+xx
ADD	HL,BC	09	11	HL<-<HL>+<BC>
ADD	HL,DE	19	11	HL<-<HL>+<DE>
ADD	HL,HL	29	11	HL<-<HL>+<HL>
ADD	HL,SP	39	11	HL<-<HL>+<SP>
ADD	IX,BC	DD 09	15	IX<-<IX>+<BC>
ADD	IX,DE	DD 19	15	IX<-<IX>+<DE>
ADD	IX,IX	DD 29	15	IX<-<IX>+<IX>
ADD	IX,SP	DD 39	15	IX<-<IX>+<SP>

ADD	IY,BC	FD 09	15	IY <– <IY> + <BC>
ADD	IY,DE	FD 19	15	IY <– <IY> + <DE>
ADD	IY,IY	FD 29	15	IY <– <IY> + <IY>
ADD	IY,SP	FD 39	15	IY <– <IY> + <SP>
AND	(HL)	A6	7	A <– <A> and <(<HL>)>
AND	(IX+zz)	DD A6 zz	19	A <– <A> and <(<IX>+zz)>
AND	(IY+zz)	FD A6 zz	19	A <– <A> and <(<IY>+zz)>
AND	A	A7	4	A <– <A> and <A>
AND	B	A0	4	A <– <A> and <B>
AND	C	A1	4	A <– <A> and <C>
AND	D	A2	4	A <– <A> and <D>
AND	E	A3	4	A <– <A> and <E>
AND	H	A4	4	A <– <A> and <H>
AND	L	A5	4	A <– <A> and <L>
AND	xx	E6 xx	7	A <– <A> and xx
BIT	0,(HL)	CB 46	12	Z <– not $<(<HL>)>_0$
BIT	0,(IX+zz)	DD CB zz 46	20	Z <– not $<(<IX>+zz)>_0$
BIT	0,(IY+zz)	FD CB zz 46	20	Z <– not $<(<IY>+zz)>_0$
BIT	0,A	CB 47	8	Z <– not $<A>_0$
BIT	0,B	CB 40	8	Z <– not $<B>_0$
BIT	0,C	CB 41	8	Z <– not $<C>_0$
BIT	0,D	CB 42	8	Z <– not $<D>_0$
BIT	0,E	CB 43	8	Z <– not $<E>_0$
BIT	0,H	CB 44	8	Z <– not $<H>_0$
BIT	0,L	CB 45	8	Z <– not $<L>_0$
BIT	1,(HL)	CB 4E	12	Z <– not $<(<HL>)>_1$
BIT	1,(IX+zz)	DD CB zz 4E	20	Z <– not $<(<IX>+zz)>_1$
BIT	1,(IY+zz)	FD CB zz 4E	20	Z <– not $<(<IY>+zz)>_1$
BIT	1,A	CB 4F	8	Z <– not $<A>_1$
BIT	1,B	CB 48	8	Z <– not $<B>_1$
BIT	1,C	CB 49	8	Z <– not $<C>_1$
BIT	1,D	CB 4A	8	Z <– not $<D>_1$
BIT	1,E	CB 4B	8	Z <– not $<E>_1$
BIT	1,H	CB 4C	8	Z <– not $<H>_1$
BIT	1,L	CB 4D	8	Z <– not $<L>_1$
BIT	2,(HL)	CB 56	12	Z <– not $<(<HL>)>_2$
BIT	2,(IX+zz)	DD CB zz 56	20	Z <– not $<(<IX>+zz)>_2$
BIT	2,(IY+zz)	FD CB zz 56	20	Z <– not $<(<IY>+zz)>_2$
BIT	2,A	CB 57	8	Z <– not $<A>_2$

BIT	2,B	CB 50	8	Z <– not <B>$_2$
BIT	2,C	CB 51	8	Z <– not <C>$_2$
BIT	2,D	CB 52	8	Z <– not <D>$_2$
BIT	2,E	CB 53	8	Z <– not <E>$_2$
BIT	2,H	CB 54	8	Z <– not <H>$_2$
BIT	2,L	CB 55	8	Z <– not <L>$_2$
BIT	3,(HL)	CB 5E	12	Z <– not <(<HL>)>$_3$
BIT	3,(IX+zz)	DD CB zz 5E	20	Z <– not <(<IX>+zz)>$_3$
BIT	3,(IY+zz)	FD CB zz 5E	20	Z <– not <(<IY>+zz)>$_3$
BIT	3,A	CB 5F	8	Z <– not <A>$_3$
BIT	3,B	CB 58	8	Z <– not <B>$_3$
BIT	3,C	CB 59	8	Z <– not <C>$_3$
BIT	3,D	CB 5A	8	Z <– not <D>$_3$
BIT	3,E	CB 5B	8	Z <– not <E>$_3$
BIT	3,H	CB 5C	8	Z <– not <H>$_3$
BIT	3,L	CB 5D	8	Z <– not <L>$_3$
BIT	4,(HL)	CB 66	12	Z <– not <(<HL>)>$_4$
BIT	4,(IX+zz)	DD CB zz 66	20	Z <– not <(<IX>+zz)>$_4$
BIT	4,(IY+zz)	FD CB zz 66	20	Z <– not <(<IY>+zz)>$_4$
BIT	4,A	CB 67	8	Z <– not <A>$_4$
BIT	4,B	CB 60	8	Z <– not <B>$_4$
BIT	4,C	CB 61	8	Z <– not <C>$_4$
BIT	4,D	CB 62	8	Z <– not <D>$_4$
BIT	4,E	CB 63	8	Z <– not <E>$_4$
BIT	4,H	CB 64	8	Z <– not <H>$_4$
BIT	4,L	CB 65	8	Z <– not <L>$_4$
BIT	5,(HL)	CB 6E	12	Z <– not <(<HL>)>$_5$
BIT	5,(IX+zz)	DD CB zz 6E	20	Z <– not <(<IX>+zz)>$_5$
BIT	5,(IY+zz)	FD CB zz 6E	20	Z <– not <(<IY>+zz)>$_5$
BIT	5,A	CB 6F	8	Z <– not <A>$_5$
BIT	5,B	CB 68	8	Z <– not <B>$_5$
BIT	5,C	CB 69	8	Z <– not <C>$_5$
BIT	5,D	CB 6A	8	Z <– not <D>$_5$
BIT	5,E	CB 6B	8	Z <– not <E>$_5$
BIT	5,H	CB 6C	8	Z <– not <H>$_5$
BIT	5,L	CB 6D	8	Z <– not <L>$_5$
BIT	6,(HL)	CB 76	12	Z <– not <(<HL>)>$_6$
BIT	6,(IX+zz)	DD CB zz 76	20	Z <– not <(<IX>+zz)>$_6$
BIT	6,(IY+zz)	FD CB zz 76	20	Z <– not <(<IY>+zz)>$_6$

BIT	6,A	CB 77	8	Z <– not <A>$_6$
BIT	6,B	CB 70	8	Z <– not <B>$_6$
BIT	6,C	CB 71	8	Z <– not <C>$_6$
BIT	6,D	CB 72	8	Z <– not <D>$_6$
BIT	6,E	CB 73	8	Z <– not <E>$_6$
BIT	6,H	CB 74	8	Z <– not <H>$_6$
BIT	6,L	CB 75	8	Z <– not <L>$_6$
BIT	7,(HL)	CB 7E	12	Z <– not <(<HL>)>$_7$
BIT	7,(IX+zz)	DD CB zz 7E	20	Z <– not <(<IX>+zz)>$_7$
BIT	7,(IY+zz)	FD CB zz 7E	20	Z <– not <(<IY>+zz)>$_7$
BIT	7,A	CB 7F	8	Z <– not <A>$_7$
BIT	7,B	CB 78	8	Z <– not <B>$_7$
BIT	7,C	CB 79	8	Z <– not <C>$_7$
BIT	7,D	CB 7A	8	Z <– not <D>$_7$
BIT	7,E	CB 7B	8	Z <– not <E>$_7$
BIT	7,H	CB 7C	8	Z <– not <H>$_7$
BIT	7,L	CB 7D	8	Z <– not <L>$_7$

CALL	C,xxyy	DC yy xx		**wenn**	<CY> = 1
			17	**dann**	SP <– <SP> – 1
					(<SP>) <– <PC>$_H$
					SP <– <SP> – 1
					(<SP>) <– <PC>$_L$
					PC <– xxyy
			10	**sonst**	keine Aktion
CALL	M,xxyy	FC yy xx		**wenn**	<S> = 1
			17	**dann**	SP <– <SP> – 1
					(<SP>) <– <PC>$_H$
					SP <– <SP> – 1
					(<SP>) <– <PC>$_L$
					PC <– xxyy
			10	**sonst**	keine Aktion
CALL	NC,xxyy	D4 yy xx		**wenn**	<CY> = 0
			17	**dann**	SP <– <SP> – 1
					(<SP>) <– <PC>$_H$
					SP <– <SP> – 1
					(<SP>) <– <PC>$_L$
					PC <– xxyy
			10	**sonst**	keine Aktion

CALL	NZ,xxyy	C4 yy xx		**wenn**	<Z> = 0
			17	**dann**	SP <– <SP> – 1
					(<SP>) <– <PC>$_H$
					SP <– <SP> – 1
					(<SP>) <– <PC>$_L$
					PC <– xxyy
			10	**sonst**	keine Aktion
CALL	P,xxyy	F4 yy xx		**wenn**	<S> = 0
			17	**dann**	SP <– <SP> – 1
					(<SP>) <– <PC>$_H$
					SP <– <SP> – 1
					(<SP>) <– <PC>$_L$
					PC <– xxyy
			10	**sonst**	keine Aktion
CALL	PE,xxyy	EC yy xx		**wenn**	<P> = 1
			17	**dann**	SP <– <SP> – 1
					(<SP>) <– <PC>$_H$
					SP <– <SP> – 1
					(<SP>) <– <PC>$_L$
					PC <– xxyy
			10	**sonst**	keine Aktion
CALL	PO,xxyy	E4 yy xx		**wenn**	<P> = 0
			17	**dann**	SP <– <SP> – 1
					(<SP>) <– <PC>$_H$
					SP <– <SP> – 1
					(<SP>) <– <PC>$_L$
					PC <– xxyy
			10	**sonst**	keine Aktion
CALL	Z,xxyy	CC yy xx		**wenn**	<Z> = 1
			17	**dann**	SP <– <SP> – 1
					(<SP>) <– <PC>$_H$
					SP <– <SP> – 1
					(<SP>) <– <PC>$_L$
					PC <– xxyy
			10	**sonst**	keine Aktion
CALL	xxyy	CD yy xx	17	SP <– <SP> – 1	
				(<SP>) <– <PC>$_H$	
				SP <– <SP> – 1	

				(<SP>) <- <PC>$_L$ PC <- xxyy
CCF		3F	4	CY <- **not** <CY>
CP	(HL)	BE	7	<A> - <(<HL>)>
CP	(IX+zz)	DD BE zz	19	<A> - <(<IX>+zz)>
CP	(IY+zz)	FD BE zz	19	<A> - <(<IY>+zz)>
CP	A	BF	4	<A> - <A>
CP	B	B8	4	<A> - <B>
CP	C	B9	4	<A> - <C>
CP	D	BA	4	<A> - <D>
CP	E	BB	4	<A> - <E>
CP	H	BC	4	<A> - <H>
CP	L	BD	4	<A> - <L>
CP	xx	FE xx	7	<A> - xx
CPD		ED A9	16	<A> - <(<HL>)> HL <- <HL> - 1 BC <- <BC> - 1
CPDR		ED B9	je 21	**wiederhole** <A> - <(<HL>)> HL <- <HL> - 1 BC <- <BC> - 1
			16	**bis** <BC> = 0 oder <Z> = 1
CPI		ED A1	16	<A> - <(<HL>)> HL <- <HL> + 1 BC <- <BC> - 1
CPIR		ED B1	je 21	**wiederhole** <A> - <(<HL>)> HL <- <HL> + 1 BC <- <BC> - 1
			16	**bis** <BC> = 0 oder <Z> = 1
CPL		2F	4	A <- **not** <A>
DAA		27	4	korrigiert <A> nach BCD-Operation
DEC	(HL)	35	11	(<HL>) <- <(<HL>)> - 1
DEC	(IX+zz)	DD 35 zz	23	(<IX>+zz) <- <(<IX>+zz)> - 1

DEC	(IY+zz)	FD 35 zz	23	(<IY>+zz) <– <(<IY>+zz)> – 1
DEC	A	3D	4	A <– <A> – 1
DEC	B	05	4	B <– <B> – 1
DEC	BC	0B	6	BC <– <BC> – 1
DEC	C	0D	4	C <– <C> – 1
DEC	D	15	4	D <– <D> – 1
DEC	DE	1B	6	DE <– <DE> – 1
DEC	E	1D	4	E <– <E> – 1
DEC	H	25	4	H <– <H> – 1
DEC	HL	2B	6	HL <– <HL> – 1
DEC	IX	DD 2B	10	IX <– <IX> – 1
DEC	IY	FD 2B	10	IY <– <IY> – 1
DEC	L	2D	4	L <– <L> – 1
DEC	SP	3B	6	SP <– <SP> – 1
DI		F3	4	IFF <– 0
DJNZ	tt	10 tt		B <– <B> – 1
				wenn <B> = 0
			8	**dann** keine Aktion
			13	**sonst** PC <– <PC> + tt
EI		FB	4	IFF <– 1 (nach folgendem Befehl)
EX	(SP),HL	E3	19	(<SP>+1) & (<SP>) & HL <– <HL> & <(<SP>+1)> & <(<SP>)>
EX	(SP),IX	DD E3	23	(<SP>+1) & (<SP>) & IX <– <IX> & <(<SP>+1)> & <(<SP>)>
EX	(SP),IY	FD E3	23	(<SP>+1) & (<SP>) & IY <– <IY> & <(<SP>+1)> & <(<SP>)>
EX	AF,AF'	08	4	AF & AF' <– <AF'> & <AF>
EX	DE,HL	EB	4	DE & HL <– <HL> & <DE>
EXX		D9	4	BC & BC' <– <BC'> & <BC> DE & DE' <– <DE'> & <DE> HL & HL' <– <HL'> & <HL>
HALT		76	4	Prozessor wird angehalten, bis Unterbrechung oder Ruecksetzen erfolgt
IM	0	ED 46	8	Unterbrechungsmodus 0 wird gesetzt
IM	1	ED 56	8	Unterbrechungsmodus 1 wird gesetzt

IM	2	ED 5E	8	Unterbrechungsmodus 2 wird gesetzt
IN	A,(C)	ED 78	12	A <– <[<BC>]>
IN	A,(xx)	DB xx	11	A <– <[<A> & xx]>
IN	B,(C)	ED 40	12	B <– <[<BC>]>
IN	C,(C)	ED 48	12	C <– <[<BC>]>
IN	D,(C)	ED 50	12	D <– <[<BC>]>
IN	E,(C)	ED 58	12	E <– <[<BC>]>
IN	H,(C)	ED 60	12	H <– <[<BC>]>
IN	L,(C)	ED 68	12	L <– <[<BC>]>
INC	(HL)	34	11	(<HL>) <– <(<HL>)> + 1
INC	(IX+zz)	DD 34 zz	23	(<IX>+zz) <– <(<IX>+zz)> + 1
INC	(IY+zz)	FD 34 zz	23	(<IY>+zz) <– <(<IY>+zz)> + 1
INC	A	3C	4	A <– <A> + 1
INC	B	04	4	B <– <B> + 1
INC	BC	03	6	BC <– <BC> + 1
INC	C	0C	4	C <– <C> + 1
INC	D	14	4	D <– <D> + 1
INC	DE	13	6	DE <– <DE> + 1
INC	E	1C	4	E <– <E> + 1
INC	H	24	4	H <– <H> + 1
INC	HL	23	6	HL <– <HL> + 1
INC	IX	DD 23	10	IX <– <IX> + 1
INC	IY	FD 23	10	IY <– <IY> + 1
INC	L	2C	4	L <– <L> + 1
INC	SP	33	6	SP <– <SP> + 1
IND		ED AA	16	(<HL>) <– <[<BC>]> B <– <B> – 1 HL <– <HL> – 1
INDR		ED BA	je 21	**wiederhole** (<HL>) <– <[<BC>]> HL <– <HL> – 1 B <– <B> – 1
			16	**bis** <B> = 0
INI		ED A2	16	(<HL>) <– <[<BC>]> B <– <B> – 1 HL <– <HL> + 1

INIR		ED B2	je 21	**wiederhole**	
					(<HL>) <– <[<BC>]>
					HL <– <HL> + 1
					B <– <B> – 1
				16	**bis** <B> = 0
JP	(HL)	E9	4	PC <– <HL>	
JP	(IX)	DD E9	8	PC <– <IX>	
JP	(IY)	FD E9	8	PC <– <IY>	
JP	C,xxyy	DA yy xx		**wenn**	<CY> = 1
			10	**dann**	PC <– xxyy
			10	**sonst**	keine Aktion
JP	M,xxyy	FA yy xx		**wenn**	<S> = 1
			10	**dann**	PC <– xxyy
			10	**sonst**	keine Aktion
JP	NC,xxyy	D2 yy xx		**wenn**	<CY> = 0
			10	**dann**	PC <– xxyy
			10	**sonst**	keine Aktion
JP	NZ,xxyy	C2 yy xx		**wenn**	<Z> = 0
			10	**dann**	PC <– xxyy
			10	**sonst**	keine Aktion
JP	P,xxyy	F2 yy xx		**wenn**	<S> = 0
			10	**dann**	PC <– xxyy
			10	**sonst**	keine Aktion
JP	PE,xxyy	EA yy xx		**wenn**	<P> = 1
			10	**dann**	PC <– xxyy
			10	**sonst**	keine Aktion
JP	PO,xxyy	E2 yy xx		**wenn**	<P> = 0
			10	**dann**	PC <– xxyy
			10	**sonst**	keine Aktion
JP	Z,xxyy	CA yy xx		**wenn**	<Z> = 1
			10	**dann**	PC <– xxyy
			10	**sonst**	keine Aktion
JP	xxyy	C3 yy xx	10	PC <– xxyy	
JR	C,tt	38 tt		**wenn**	<CY> = 1
			12	**dann**	PC <– <PC> + tt
			7	**sonst**	keine Aktion
JR	NC,tt	30 tt		**wenn**	<CY> = 0
			12	**dann**	PC <– <PC> + tt
			7	**sonst**	keine Aktion
JR	NZ,tt	20 tt		**wenn**	<Z> = 0
			12	**dann**	PC <– <PC> + tt
			7	**sonst**	keine Aktion

JR	Z,tt	28 tt		**wenn** <Z> = 1
			12	**dann** PC <– <PC> + tt
			7	**sonst** keine Aktion
JR	tt 18	tt 12		PC <– <PC> + tt
LD	(BC),A	02	7	(<BC>) <– <A>
LD	(DE),A	12	7	(<DE>) <– <A>
LD	(HL),A	77	7	(<HL>) <– <A>
LD	(HL),B	70	7	(<HL>) <– <B>
LD	(HL),C	71	7	(<HL>) <– <C>
LD	(HL),D	72	7	(<HL>) <– <D>
LD	(HL),E	73	7	(<HL>) <– <E>
LD	(HL),H	74	7	(<HL>) <– <H>
LD	(HL),L	75	7	(<HL>) <– <L>
LD	(HL),xx	36 xx	10	(<HL>) <– xx
LD	(IX+zz),A	DD 77 zz	19	(<IX>+zz) <– <A>
LD	(IX+zz),B	DD 70 zz	19	(<IX>+zz) <– <B>
LD	(IX+zz),C	DD 71 zz	19	(<IX>+zz) <– <C>
LD	(IX+zz),D	DD 72 zz	19	(<IX>+zz) <– <D>
LD	(IX+zz),E	DD 73 zz	19	(<IX>+zz) <– <E>
LD	(IX+zz),H	DD 74 zz	19	(<IX>+zz) <– <H>
LD	(IX+zz),L	DD 75 zz	19	(<IX>+zz) <– <L>
LD	(IX+zz),xx	DD 36 zz xx	19	(<IX>+zz) <– xx
LD	(IY+zz),A	FD 77 zz	19	(<IY>+zz) <– <A>
LD	(IY+zz),B	FD 70 zz	19	(<IY>+zz) <– <B>
LD	(IY+zz),C	FD 71 zz	19	(<IY>+zz) <– <C>
LD	(IY+zz),D	FD 72 zz	19	(<IY>+zz) <– <D>
LD	(IY+zz),E	FD 73 zz	19	(<IY>+zz) <– <E>
LD	(IY+zz),H	FD 74 zz	19	(<IY>+zz) <– <H>
LD	(IY+zz),L	FD 75 zz	19	(<IY>+zz) <– <L>
LD	(IY+zz),xx	FD 36 zz xx	19	(<IY>+zz) <– xx
LD	(xxyy),A	32 yy xx	13	(xxyy) <– <A>
LD	(xxyy),BC	ED 43 yy xx	20	(xxyy) <– <C>
				(xxyy+1) <– <B>
LD	(xxyy),DE	ED 53 yy xx	20	(xxyy) <– <E>
				(xxyy+1) <– <D>
LD	(xxyy),HL	22 yy xx	16	(xxyy) <– <L>
				(xxyy+1) <– <H>
LD	(xxyy),HL	ED 63 yy xx	20	(xxyy) <– <L>
				(xxyy+1) <– <H>
LD	(xxyy),IX	DD 22 yy xx	20	(xxyy) <– <IX>L
				(xxyy+1) <– <IX>H

LD	(xxyy),IY	FD 22 yy xx	20	(xxyy) <– <IY>L (xxyy+1) <– <IY>H
LD	(xxyy),SP	ED 73 yy xx	20	(xxyy) <– <SP>L (xxyy+1) <– <SP>H
LD	A,(BC)	0A	7	A <– <(<BC>)>
LD	A,(DE)	1A	7	A <– <(<DE>)>
LD	A,(HL)	7E	7	A <– <(<HL>)>
LD	A,(IX+zz)	DD 7E zz	19	A <– <(<IX>+zz)>
LD	A,(IY+zz)	FD 7E zz	19	A <– <(<IY>+zz)>
LD	A,(xxyy)	3A yy xx	13	A <– <(xxyy)>
LD	A,A	7F	4	A <– <A>
LD	A,B	78	4	A <– <B>
LD	A,C	79	4	A <– <C>
LD	A,D	7A	4	A <– <D>
LD	A,E	7B	4	A <– <E>
LD	A,H	7C	4	A <– <H>
LD	A,I	ED 57	9	A <– <I>
LD	A,L	7D	4	A <– <L>
LD	A,R	ED 5F	9	A <– <R>
LD	A,xx	3E xx	7	A <– xx
LD	B,(HL)	46	7	B <– <(<HL>)>
LD	B,(IX+zz)	DD 46 zz	19	B <– <(<IX>+zz)>
LD	B,(IY+zz)	FD 46 zz	19	B <– <(<IY>+zz)>
LD	B,A	47	4	B <– <A>
LD	B,B	40	4	B <– <B>
LD	B,C	41	4	B <– <C>
LD	B,D	42	4	B <– <D>
LD	B,E	43	4	B <– <E>
LD	B,H	44	4	B <– <H>
LD	B,L	45	4	B <– <L>
LD	B,xx	06 xx	7	B <– xx
LD	BC,(xxyy)	ED 4B yy xx	20	BC <– <(xxyy)>
LD	BC,xxyy	01 yy xx	10	BC <– xxyy
LD	C,(HL)	4E	7	C <– <(<HL>)>
LD	C,(IX+zz)	DD 4E zz	19	C <– <(<IX>+zz)>
LD	C,(IY+zz)	FD 4E zz	19	C <– <(<IY>+zz)>
LD	C,A	4F	4	C <– <A>
LD	C,B	48	4	C <– <B>
LD	C,C	49	4	C <– <C>
LD	C,D	4A	4	C <– <D>
LD	C,E	4B	4	C <– <E>
LD	C,H	4C	4	C <– <H>
LD	C,L	4D	4	C <– <L>

LD	C,xx	0E xx	7	C <– xx
LD	D,(HL)	56	7	D <– <(<HL>)>
LD	D,(IX+zz)	DD 56 zz	19	D <– <(<IX>+zz)>
LD	D,(IY+zz)	FD 56 zz	19	D <– <(<IY>+zz)>
LD	D,A	57	4	D <– <A>
LD	D,B	50	4	D <– <B>
LD	D,C	51	4	D <– <C>
LD	D,D	52	4	D <– <D>
LD	D,E	53	4	D <– <E>
LD	D,H	54	4	D <– <H>
LD	D,L	55	4	D <– <L>
LD	D,xx	16 xx	7	D <– xx
LD	DE,(xxyy)	ED 5B yy xx	20	DE <– <(xxyy)>
LD	DE,xxyy	11 yy xx	10	DE <– xxyy
LD	E,(HL)	5E	7	E <– <(<HL>)>
LD	E,(IX+zz)	DD 5E zz	19	E <– <(<IX>+zz)>
LD	E,(IY+zz)	FD 5E zz	19	E <– <(<IY>+zz)>
LD	E,A	5F	4	E <– <A>
LD	E,B	58	4	E <– <B>
LD	E,C	59	4	E <– <C>
LD	E,D	5A	4	E <– <D>
LD	E,E	5B	4	E <– <E>
LD	E,H	5C	4	E <– <H>
LD	E,L	5D	4	E <– <L>
LD	E,xx	1E xx	7	E <– xx
LD	H,(HL)	66	7	H <– <(<HL>)>
LD	H,(IX+zz)	DD 66 zz	19	H <– <(<IX>+zz)>
LD	H,(IY+zz)	FD 66 zz	19	H <– <(<IY>+zz)>
LD	H,A	67	4	H <– <A>
LD	H,B	60	4	H <– <B>
LD	H,C	61	4	H <– <C>
LD	H,D	62	4	H <– <D>
LD	H,E	63	4	H <– <E>
LD	H,H	64	4	H <– <H>
LD	H,L	65	4	H <– <L>
LD	H,xx	26 xx	7	H <– xx
LD	HL,(xxyy)	2A yy xx	16	HL <– <(xxyy)>
LD	HL,(xxyy)	ED 6B yy xx	20	HL <– <(xxyy)>
LD	HL,xxyy	21 yy xx	10	HL <– xxyy
LD	I,A	ED 47	9	I <– <A>
LD	IX,(xxyy)	DD 2A yy xx	20	IX <– <(xxyy)>
LD	IX,xxyy	DD 21 yy xx	14	IX <– xxyy
LD	IY,(xxyy)	FD 2A yy xx	20	IY <– <(xxyy)>

LD	IY,xxyy	FD 21 yy xx	14	IY <– xxyy
LD	L,(HL)	6E	7	L <– <(<HL>)>
LD	L,(IX+zz)	DD 6E zz	19	L <– <(<IX>+zz)>
LD	L,(IY+zz)	FD 6E zz	19	L <– <(<IY>+zz)>
LD	L,A	6F	4	L <– <A>
LD	L,B	68	4	L <– <B>
LD	L,C	69	4	L <– <C>
LD	L,D	6A	4	L <– <D>
LD	L,E	6B	4	L <– <E>
LD	L,H	6C	4	L <– <H>
LD	L,L	6D	4	L <– <L>
LD	L,xx	2E xx	7	L <– xx
LD	R,A	ED 4F	9	R <– <A>
LD	SP,(xxyy)	ED 7B yy xx	20	SP <– <(xxyy)>
LD	SP,HL	F9	6	SP <– <HL>
LD	SP,IX	DD F9	10	SP <– <IX>
LD	SP,IY	FD F9	10	SP <– <IY>
LD	SP,xxyy	31 yy xx	10	SP <– xxyy
LDD		ED A8	16	(<DE>) <– <(<HL>)> DE <– <DE> – 1 HL <– <HL> – 1 BC <– <BC> – 1
LDDR		ED B8	je 21	**wiederhole** (<DE>) <– <(<HL>)> <DE> – 1 HL <– <HL> – 1 BC <– <BC> – 1
			16	**bis** <BC> = 0
LDI		ED A0	16	(<DE>) <– <(<HL>)> DE <– <DE> + 1 HL <– <HL> + 1 BC <– <BC> – 1
LDIR		ED B0	je 21	**wiederhole** (<DE>) <– <(<HL>)> DE <– <DE> + 1 HL <– <HL> + 1 BC <– <BC> – 1
			16	**bis** <BC> = 0

NEG		ED 44	8	A <– 0 – <A>
NOP		00	4	keine Aktion
OR	(HL)	B6	7	A <– <A> **or** <(<HL>)>
OR	(IX+zz)	DD B6 zz	19	A <– <A> **or** <(<IX>+zz)>
OR	(IY+zz)	FD B6 zz	19	A <– <A> **or** <(<IY>+zz)>
OR	A	B7	4	A <– <A> **or** <A>
OR	B	B0	4	A <– <A> **or** <B>
OR	C	B1	4	A <– <A> **or** <C>
OR	D	B2	4	A <– <A> **or** <D>
OR	E	B3	4	A <– <A> **or** <E>
OR	H	B4	4	A <– <A> **or** <H>
OR	L	B5	4	A <– <A> **or** <L>
OR	xx	F6 xx	7	A <– <A> **or** xx
OTDR		ED BB		je 21 **wiederhole**
				[<BC>] <– <(<HL>)>
				HL <– <HL> – 1
				B <– <B> – 1
			16	**bis** <B> = 0
OTIR		ED B3		je 21 **wiederhole**
				[<BC>] <– <(<HL>)>
				HL <– <HL> + 1
				B <– <B> – 1
			16	**bis** <B> = 0
OUT	(C),A	ED 79	12	[<BC>] <– <A>
OUT	(C),B	ED 41	12	[<BC>] <– <B>
OUT	(C),C	ED 49	12	[<BC>] <– <C>
OUT	(C),D	ED 51	12	[<BC>] <– <D>
OUT	(C),E	ED 59	12	[<BC>] <– <E>
OUT	(C),H	ED 61	12	[<BC>] <– <H>
OUT	(C),L	ED 69	12	[<BC>] <– <L>
OUT	(xx),A	D3 xx	11	[<A> & xx] <– <A>
OUTD		ED AB	16	[<BC>] <– <(<HL>)>
				HL <– <HL> – 1
				B <– <B> – 1
OUTI		ED A3	16	[<BC>] <– <(<HL>)>
				HL <– <HL> + 1
				B <– <B> – 1

POP	AF	F1	10	F <- <(<SP>)> SP <- <SP> + 1 A <- <(<SP>)> SP <- <SP> + 1
POP	BC	C1	10	C <- <(<SP>)> SP <- <SP> + 1 B <- <(<SP>)> SP <- <SP> + 1
POP	DE	D1	10	E <- <(<SP>)> SP <- <SP> + 1 D <- <(<SP>)> SP <- <SP> + 1
POP	HL	E1	10	L <- <(<SP>)> SP <- <SP> + 1 H <- <(<SP>)> SP <- <SP> + 1
POP	IX	DD E1	14	IX_L <- <(<SP>)> SP <- <SP> + 1 IX_H <- <(<SP>)> SP <- <SP> + 1
POP	IY	FD E1	14	IY_L <- <(<SP>)> SP <- <SP> + 1 IY_H <- <(<SP>)> SP <- <SP> + 1
PUSH	AF	F5	11	SP <- <SP> - 1 (<SP>) <- <A> SP <- <SP> - 1 (<SP>) <- <F>
PUSH	BC	C5	11	SP <- <SP> - 1 (<SP>) <- <B> SP <- <SP> - 1 (<SP>) <- <C>
PUSH	DE	D5	11	SP <- <SP> - 1 (<SP>) <- <D> SP <- <SP> - 1 (<SP>) <- <E>
PUSH	HL	E5	11	SP <- <SP> - 1 (<SP>) <- <H> SP <- <SP> - 1 (<SP>) <- <L>
PUSH	IX	DD E5	15	SP <- <SP> - 1 (<SP>) <- $<IX>_H$

				SP <– <SP> – 1
				(<SP>) <– <IX>$_L$
PUSH	IY	FD E5	15	SP <– <SP> – 1
				(<SP>) <– <IY>$_H$
				SP <– <SP> – 1
				(<SP>) <– <IY>$_L$
RES	0,(HL)	CB 86	15	(<HL>)$_0$ <– 0
RES	0,(IX+zz)	DD CB zz 86	23	(<IX>+zz)$_0$ <– 0
RES	0,(IY+zz)	FD CB zz 86	23	(<IY>+zz)$_0$ <– 0
RES	0,A	CB 87	8	A_0 <– 0
RES	0,B	CB 80	8	B_0 <– 0
RES	0,C	CB 81	8	C_0 <– 0
RES	0,D	CB 82	8	D_0 <– 0
RES	0,E	CB 83	8	E_0 <– 0
RES	0,H	CB 84	8	H_0 <– 0
RES	0,L	CB 85	8	L_0 <– 0
RES	1,(HL)	CB 8E	15	(<HL>)$_1$ <– 0
RES	1,(IX+zz)	DD CB zz 8E	23	(<IX>+zz)$_1$ <– 0
RES	1,(IY+zz)	FD CB zz 8E	23	(<IY>+zz)$_1$ <– 0
RES	1,A	CB 8F	8	A_1 <– 0
RES	1,B	CB 88	8	B_1 <– 0
RES	1,C	CB 89	8	C_1 <– 0
RES	1,D	CB 8A	8	D_1 <– 0
RES	1,E	CB 8B	8	E_1 <– 0
RES	1,H	CB 8C	8	H_1 <– 0
RES	1,L	CB 8D	8	L_1 <– 0
RES	2,(HL)	CB 96	15	(<HL>)$_2$ <– 0
RES	2,(IX+zz)	DD CB zz 96	23	(<IX>+zz)$_2$ <– 0
RES	2,(IY+zz)	FD CB zz 96	23	(<IY>+zz)$_2$ <– 0
RES	2,A	CB 97	8	A_2 <– 0
RES	2,B	CB 90	8	B_2 <– 0
RES	2,C	CB 91	8	C_2 <– 0
RES	2,D	CB 92	8	D_2 <– 0
RES	2,E	CB 93	8	E_2 <– 0
RES	2,H	CB 94	8	H_2 <– 0
RES	2,L	CB 95	8	L_2 <– 0
RES	3,(HL)	CB 9E	15	(<HL>)$_3$ <– 0
RES	3,(IX+zz)	DD CB zz 9E	23	(<IX>+zz)$_3$ <– 0
RES	3,(IY+zz)	FD CB zz 9E	23	(<IY>+zz)$_3$ <– 0

RES	3,A	CB 9F	8	$A_3 <- 0$
RES	3,B	CB 98	8	$B_3 <- 0$
RES	3,C	CB 99	8	$C_3 <- 0$
RES	3,D	CB 9A	8	$D_3 <- 0$
RES	3,E	CB 9B	8	$E_3 <- 0$
RES	3,H	CB 9C	8	$H_3 <- 0$
RES	3,L	CB 9D	8	$L_3 <- 0$
RES	4,(HL)	CB A6	15	$(<HL>)_4 <- 0$
RES	4,(IX+zz)	DD CB zz A6	23	$(<IX>+zz)_4 <- 0$
RES	4,(IY+zz)	FD CB zz A6	23	$(<IY>+zz)_4 <- 0$
RES	4,A	CB A7	8	$A_4 <- 0$
RES	4,B	CB A0	8	$B_4 <- 0$
RES	4,C	CB A1	8	$C_4 <- 0$
RES	4,D	CB A2	8	$D_4 <- 0$
RES	4,E	CB A3	8	$E_4 <- 0$
RES	4,H	CB A4	8	$H_4 <- 0$
RES	4,L	CB A5	8	$L_4 <- 0$
RES	5,(HL)	CB AE	15	$(<HL>)_5 <- 0$
RES	5,(IX+zz)	DD CB zz AE	23	$(<IX>+zz)_5 <- 0$
RES	5,(IY+zz)	FD CB zz AE	23	$(<IY>+zz)_5 <- 0$
RES	5,A	CB AF	8	$A_5 <- 0$
RES	5,B	CB A8	8	$B_5 <- 0$
RES	5,C	CB A9	8	$C_5 <- 0$
RES	5,D	CB AA	8	$D_5 <- 0$
RES	5,E	CB AB	8	$E_5 <- 0$
RES	5,H	CB AC	8	$H_5 <- 0$
RES	5,L	CB AD	8	$L_5 <- 0$
RES	6,(HL)	CB B6	15	$(<HL>)_6 <- 0$
RES	6,(IX+zz)	DD CB zz B6	23	$(<IX>+zz)_6 <- 0$
RES	6,(IY+zz)	FD CB zz B6	23	$(<IY>+zz)_6 <- 0$
RES	6,A	CB B7	8	$A_6 <- 0$
RES	6,B	CB B0	8	$B_6 <- 0$
RES	6,C	CB B1	8	$C_6 <- 0$
RES	6,D	CB B2	8	$D_6 <- 0$
RES	6,E	CB B3	8	$E_6 <- 0$
RES	6,H	CB B4	8	$H_6 <- 0$
RES	6,L	CB B5	8	$L_6 <- 0$
RES	7,(HL)	CB BE	15	$(<HL>)_7 <- 0$
RES	7,(IX+zz)	DD CB zz BE	23	$(<IX>+zz)_7 <- 0$

RES	7,(IY+zz)	FD CB zz BE	23	$(<IY>+zz)_7 <- 0$
RES	7,A	CB BF	8	$A_7 <- 0$
RES	7,B	CB B8	8	$B_7 <- 0$
RES	7,C	CB B9	8	$C_7 <- 0$
RES	7,D	CB BA	8	$D_7 <- 0$
RES	7,E	CB BB	8	$E_7 <- 0$
RES	7,H	CB BC	8	$H_7 <- 0$
RES	7,L	CB BD	8	$L_7 <- 0$
RET		C9	10	$PC_L <- <(<SP>)>$ $SP <- <SP> + 1$ $PC_H <- <(<SP>)>$ $SP <- <SP> + 1$
RET	C	D8		**wenn** $<CY> = 1$
			11	**dann** $PC_L <- <(<SP>)>$ $SP <- <SP> + 1$ $PC_H <- <(<SP>)>$ $SP <- <SP> + 1$
			5	**sonst** keine Aktion
RET	M	F8		**wenn** $<S> = 1$
			11	**dann** $PC_L <- <(<SP>)>$ $SP <- <SP> + 1$ $PC_H <- <(<SP>)>$ $SP <- <SP> + 1$
			5	**sonst** keine Aktion
RET	NC	D0		**wenn** $<CY> = 0$
			11	**dann** $PC_L <- <(<SP>)>$ $SP <- <SP> + 1$ $PC_H <- <(<SP>)>$ $SP <- <SP> + 1$
			5	**sonst** keine Aktion
RET	NZ	D0		**wenn** $<Z> = 0$
			11	**dann** $PC_L <- <(<SP>)>$ $SP <- <SP> + 1$ $PC_H <- <(<SP>)>$ $SP <- <SP> + 1$
			5	**sonst** keine Aktion
RET	P	F0		**wenn** $<CY> = 0$
			11	**dann** $PC_L <- <(<SP>)>$ $SP <- <SP> + 1$ $PC_H <- <(<SP>)>$ $SP <- <SP> + 1$
			5	**sonst** keine Aktion

RET	PE	E8		**wenn**	<P> = 1
			11	**dann**	PC_L <– <(<SP>)>
					SP <– <SP> + 1
					PC_H <– <(<SP>)>
					SP <– <SP> + 1
			5	**sonst**	keine Aktion
RET	PO	E0		**wenn**	<P> = 0
			11	**dann**	PC_L <– <(<SP>)>
					SP <– <SP> + 1
					PC_H <– <(<SP>)>
					SP <– <SP> + 1
			5	**sonst**	keine Aktion
RET	Z	C8		**wenn**	<Z> = 1
			11	**dann**	PC_L <– <(<SP>)>
					SP <– <SP> + 1
					PCH <– <(<SP>)>
					SP <– <SP> + 1
			5	**sonst**	keine Aktion
RETI		ED 4D	14		PC_L <– <(<SP>)>
					SP <– <SP> + 1
					PC_H <– <(<SP>)>
					SP <– <SP> + 1
					Signalisieren des Endes einer Unterbrechungsroutine
RETN		ED 45	14		PC_L <– <(<SP>)>
					SP <– <SP> + 1
					PC_H <– <(<SP>)>
					SP <– <SP> + 1
					IFF1 <– <IFF2>
					Ende einer nicht maskierbaren Unterbrechungsroutine
RL	(HL)	CB 16	15		CY & $(<HL>)_{7,\ldots,0}$ <– $<(<HL>)>_{7,\ldots,0}$ & <CY>
RL	(IX+zz)	DD CB zz 16	23		CY & $(<IX>+zz)_{7,\ldots,0}$ <– $<(<IX>+zz)>_{7,\ldots,0}$ & <CY>
RL	(IY+zz)	FD CB zz 16	23		CY & $(<IY>+zz)_{7,\ldots,0}$ <– $<(<IY>+zz)>_{7,\ldots,0}$ & <CY>
RL	A	CB 17	8		CY & $A_{7,\ldots,0}$ <– $<A>_{7,\ldots,0}$ & <CY>

RL	B	CB 10	8	CY & $B_{7,...,0}$ <– <B>$_{7,...,0}$ & <CY>
RL	C	CB 11	8	CY & $C_{7,...,0}$ <– <C>$_{7,...,0}$ & <CY>
RL	D	CB 12	8	CY & $D_{7,...,0}$ <– <D>$_{7,...,0}$ & <CY>
RL	E	CB 13	8	CY & $E_{7,...,0}$ <– <E>$_{7,...,0}$ & <CY>
RL	H	CB 14	8	CY & $H_{7,...,0}$ <– <H>$_{7,...,0}$ & <CY>
RL	L	CB 15	8	CY & $L_{7,...,0}$ <– <L>$_{7,...,0}$ & <CY>
RLA		17	4	CY & $A_{7,...,0}$ <– <A>$_{7,...,0}$ & <CY>
RLC	(HL)	CB 06	15	CY <– <(<HL>)>$_7$ (<HL>)$_{7,...,1}$ <– <(<HL>)>$_{6,...,0}$ (<HL>)$_0$ <– <CY>
RLC	(IX+zz)	DD CB zz 06	23	CY <– <(<IX>+zz)>$_7$ (<IX>+zz)$_{7,...,1}$ <– <(<IX>+zz)>$_{6,...,0}$ (<IX>+zz)$_0$ <– <CY>
RLC	(IY+zz)	FD CB zz 06	23	CY <– <(<IY>+zz)>$_7$ (<IY>+zz)$_{7,...,1}$ <– <(<IY>+zz)>$_{6,...,0}$ (<IY>+zz)$_0$ <– <CY>
RLC	A	CB 07	8	CY <– <A>$_7$ $A_{7,...,1}$ <– <A>$_{6,...,0}$ A_0 <– <CY>
RLC	B	CB 00	8	CY <– <B>$_7$ $B_{7,...,1}$ <– <B>$_{6,...,0}$ B_0 <– <CY>
RLC	C	CB 01	8	CY <– <C>$_7$ $C_{7,...,1}$ <– <C>$_{6,...,0}$ C_0 <– <CY>
RLC	D	CB 02	8	CY <– <D>$_7$ $D_{7,...,1}$ <– <D>$_{6,...,0}$ D_0 <– <CY>
RLC	E	CB 03	8	CY <– <E>$_7$ $E_{7,...,1}$ <– <E>$_{6,...,0}$ E_0 <– <CY>
RLC	H	CB 04	8	CY <– <H>$_7$ $H_{7,...,1}$ <– <H>$_{6,...,0}$ H_0 <– <CY>

RLC	L	CB 05	8	CY <– <L>$_7$ L$_{7,...,1}$ <– <L>$_{6,...,0}$ L$_0$ <– <CY>
RLCA		07	4	CY <– <A>$_7$ A$_{7,...,1}$ <– <A>$_{6,...,0}$ A$_0$ <– <CY>
RLD		ED 6F	18	A$_{3,...,0}$ & (<HL>)$_{7,...,0}$ <– <(<HL>)>$_{7,...,0}$ & <A>$_{3,...,0}$
RR	(HL)	CB 1E	15	CY & (<HL>)$_{0,...,7}$ <– <(<HL>)>$_{0,...,7}$ & <CY>
RR	(IX+zz)	DD CB zz 1E	23	CY & (<IX>+zz)$_{0,...,7}$ <– <(<IX>+zz)>$_{0,...,7}$ & <CY>
RR	(IY+zz)	FD CB zz 1E	23	CY & (<IY>+zz)$_{0,...,7}$ <– <(<IY>+zz)>$_{0,...,7}$ & <CY>
RR	A	CB 1F	8	CY & A$_{0,...,7}$ <– <A>$_{0,...,7}$ & <CY>
RR	B	CB 18	8	CY & B$_{0,...,7}$ <– <B>$_{0,...,7}$ & <CY>
RR	C	CB 19	8	CY & C$_{0,...,7}$ <– <C>$_{0,...,7}$ & <CY>
RR	D	CB 1A	8	CY & D$_{0,...,7}$ <– <D>$_{0,...,7}$ & <CY>
RR	E	CB 1B	8	CY & E$_{0,...,7}$ <– <E>$_{0,...,7}$ & <CY>
RR	H	CB 1C	8	CY & H$_{0,...,7}$ <– <H>$_{0,...,7}$ & <CY>
RR	L	CB 1D	8	CY & L$_{0,...,7}$ <– <L>$_{0,...,7}$ & <CY>
RRA		1F	4	CY & A$_{0,...,7}$ <– <A>$_{0,...,7}$ & <CY>
RRC	(HL)	CB 0E	15	CY <– <(<HL>)>$_0$ (<HL>)$_{0,...,6}$ <– <(<HL>)>$_{1,...,7}$ (<HL>)$_7$ <– <CY>
RRC	(IX+zz)	DD CB zz 0E	23	CY <– <(<IX>+zz)>$_0$ (<IX>+zz)$_{0,...,6}$ <– <(<IX>+zz)>$_{1,...,7}$ (<IX>+zz)$_7$ <– <CY>
RRC	(IY+zz)	FD CB zz 0E	23	CY <– <(<IY>+zz)>$_0$ (<IY>+zz)$_{0,...,6}$ <– <(<IY>+zz)>$_{1,...,7}$ (<IY>+zz)$_7$ <– <CY>

RRC	A	CB 0F	8	$CY \leftarrow <A>_0$ $A_{0,...,6} \leftarrow <A>_{1,...,7}$ $A_7 \leftarrow <CY>$
RRC	B	CB 08	8	$CY \leftarrow <B>_0$ $B_{0,...,6} \leftarrow <B>_{1,...,7}$ $B_7 \leftarrow <CY>$
RRC	C	CB 09	8	$CY \leftarrow <C>_0$ $C_{0,...,6} \leftarrow <C>_{1,...,7}$ $C_7 \leftarrow <CY>$
RRC	D	CB 0A	8	$CY \leftarrow <D>_0$ $D_{0,...,6} \leftarrow <D>_{1,...,7}$ $D_7 \leftarrow <CY>$
RRC	E	CB 0B	8	$CY \leftarrow <E>_0$ $E_{0,...,6} \leftarrow <E>_{1,...,7}$ $E_7 \leftarrow <CY>$
RRC	H	CB 0C	8	$CY \leftarrow <H>_0$ $H_{0,...,6} \leftarrow <H>_{1,...,7}$ $H_7 \leftarrow <CY>$
RRC	L	CB 0D	8	$CY \leftarrow <L>_0$ $L_{0,...,6} \leftarrow <L>_{1,...,7}$ $L_7 \leftarrow <CY>$
RRCA		0F	4	$CY \leftarrow <A>_0$ $A_{0,...,6} \leftarrow <A>_{1,...,7}$ $A_7 \leftarrow <CY>$
RRD		ED 67	18	$(<HL>)_{0,...,7}$ & $A_{0,...,3} \leftarrow$ $<(<HL>)>_{4,...,7}$ & $<A>_{0,...,3}$ & $<(<HL>)>_{0,...,3}$
RST	00H	C7	11	$SP \leftarrow <SP> - 1$ $(<SP>) \leftarrow <PC>_H$ $SP \leftarrow <SP> - 1$ $(<SP>) \leftarrow <PC>_L$ $PC \leftarrow 0000H$
RST	08H	CF	11	$SP \leftarrow <SP> - 1$ $(<SP>) \leftarrow <PC>_H$ $SP \leftarrow <SP> - 1$ $(<SP>) \leftarrow <PC>_L$ $PC \leftarrow 0008H$
RST	10H	D7	11	$SP \leftarrow <SP> - 1$ $(<SP>) \leftarrow <PC>_H$ $SP \leftarrow <SP> - 1$

				(<SP>) <- <PC>$_L$
				PC <- 0010H
RST	18H	DF	11	SP <- <SP> - 1
				(<SP>) <- <PC>$_H$
				SP <- <SP> - 1
				(<SP>) <- <PC>$_L$
				PC <- 0018H
RST	20H	E7	11	SP <- <SP> - 1
				(<SP>) <- <PC>$_H$
				SP <- <SP> - 1
				(<SP>) <- <PC>$_L$
				PC <- 0020H
RST	28H	EF	11	SP <- <SP> - 1
				(<SP>) <- <PC>$_H$
				SP <- <SP> - 1
				(<SP>) <- <PC>$_L$
				PC <- 0028H
RST	30H	F7	11	SP <- <SP> - 1
				(<SP>) <- <PC>$_H$
				SP <- <SP> - 1
				(<SP>) <- <PC>$_L$
				PC <- 0030H
RST	38H	FF	11	SP <- <SP> - 1
				(<SP>) <- <PC>$_H$
				SP <- <SP> - 1
				(<SP>) <- <PC>$_L$
				PC <- 0038H
SBC	A,(HL)	9E	7	A <- <A> - <(<HL>)> - <CY>
SBC	A,(IX+zz)	DD 9E zz	19	A <- <A> - <(<IX>+zz)> - <CY>
SBC	A,(IY+zz)	FD 9E zz	19	A <- <A> - <(<IY>+zz)> - <CY>
SBC	A,A	9F	4	A <- <A> - <A> - <CY>
SBC	A,B	98	4	A <- <A> - <B> - <CY>
SBC	A,C	99	4	A <- <A> - <C> - <CY>
SBC	A,D	9A	4	A <- <A> - <D> - <CY>
SBC	A,E	9B	4	A <- <A> - <E> - <CY>
SBC	A,H	9C	4	A <- <A> - <H> - <CY>
SBC	A,L	9D	4	A <- <A> - <L> - <CY>
SBC	A,xx	DE xx	7	A <- <A> - xx - <CY>
SBC	HL,BC	ED 42	15	HL <- <HL> - <BC> - <CY>
SBC	HL,DE	ED 52	15	HL <- <HL> - <DE> - <CY>
SBC	HL,HL	ED 62	15	HL <- <HL> - <HL> - <CY>
SBC	HL,SP	ED 72	15	HL <- <HL> - <SP> - <CY>

SCF		37	4	CY <– 1
SET	0,(HL)	CB C6	15	(<HL>)$_0$ <– 1
SET	0,(IX+zz)	DD CB zz C6	23	(<IX>+zz)$_0$ <– 1
SET	0,(IY+zz)	FD CB zz C6	23	(<IY>+zz)$_0$ <– 1
SET	0,A	CB C7	8	A$_0$ <– 1
SET	0,B	CB C0	8	B$_0$ <– 1
SET	0,C	CB C1	8	C$_0$ <– 1
SET	0,D	CB C2	8	D$_0$ <– 1
SET	0,E	CB C3	8	E$_0$ <– 1
SET	0,H	CB C4	8	H$_0$ <– 1
SET	0,L	CB C5	8	L$_0$ <– 1
SET	1,(HL)	CB CE	15	(<HL>)$_1$ <– 1
SET	1,(IX+zz)	DD CB zz CE	23	(<IX>+zz)$_1$ <– 1
SET	1,(IY+zz)	FD CB zz CE	23	(<IY>+zz)$_1$ <– 1
SET	1,A	CB CF	8	A$_1$ <– 1
SET	1,B	CB C8	8	B$_1$ <– 1
SET	1,C	CB C9	8	C$_1$ <– 1
SET	1,D	CB CA	8	D$_1$ <– 1
SET	1,E	CB CB	8	E$_1$ <– 1
SET	1,H	CB CC	8	H$_1$ <– 1
SET	1,L	CB CD	8	L$_1$ <– 1
SET	2,(HL)	CB D6	15	(<HL>)$_2$ <– 1
SET	2,(IX+zz)	DD CB zz D6	23	(<IX>+zz)$_2$ <– 1
SET	2,(IY+zz)	FD CB zz D6	23	(<IY>+zz)$_2$ <– 1
SET	2,A	CB D7	8	A$_2$ <– 1
SET	2,B	CB D0	8	B$_2$ <– 1
SET	2,C	CB D1	8	C$_2$ <– 1
SET	2,D	CB D2	8	D$_2$ <– 1
SET	2,E	CB D3	8	E$_2$ <– 1
SET	2,H	CB D4	8	H$_2$ <– 1
SET	2,L	CB D5	8	L$_2$ <– 1
SET	3,(HL)	CB DE	15	(<HL>)$_3$ <– 1
SET	3,(IX+zz)	DD CB zz DE	23	(<IX>+zz)$_3$ <– 1
SET	3,(IY+zz)	FD CB zz DE	23	(<IY>+zz)$_3$ <– 1
SET	3,A	CB DF	8	A$_3$ <– 1
SET	3,B	CB D8	8	B$_3$ <– 1
SET	3,C	CB D9	8	C$_3$ <– 1
SET	3,D	CB DA	8	D$_3$ <– 1
SET	3,E	CB DB	8	E$_3$ <– 1

SET	3,H	CB DC	8	$H_3 <- 1$
SET	3,L	CB DD	8	$L_3 <- 1$
SET	4,(HL)	CB E6	15	$(<HL>)_4 <- 1$
SET	4,(IX+zz)	DD CB zz E6	23	$(<IX>+zz)_4 <- 1$
SET	4,(IY+zz)	FD CB zz E6	23	$(<IY>+zz)_4 <- 1$
SET	4,A	CB E7	8	$A_4 <- 1$
SET	4,B	CB E0	8	$B_4 <- 1$
SET	4,C	CB E1	8	$C_4 <- 1$
SET	4,D	CB E2	8	$D_4 <- 1$
SET	4,E	CB E3	8	$E_4 <- 1$
SET	4,H	CB E4	8	$H_4 <- 1$
SET	4,L	CB E5	8	$L_4 <- 1$
SET	5,(HL)	CB EE	15	$(<HL>)_5 <- 1$
SET	5,(IX+zz)	DD CB zz EE	23	$(<IX>+zz)_5 <- 1$
SET	5,(IY+zz)	FD CB zz EE	23	$(<IY>+zz)_5 <- 1$
SET	5,A	CB EF	8	$A_5 <- 1$
SET	5,B	CB E8	8	$B_5 <- 1$
SET	5,C	CB E9	8	$C_5 <- 1$
SET	5,D	CB EA	8	$D_5 <- 1$
SET	5,E	CB EB	8	$E_5 <- 1$
SET	5,H	CB EC	8	$H_5 <- 1$
SET	5,L	CB ED	8	$L_5 <- 1$
SET	6,(HL)	CB F6	15	$(<HL>)_6 <- 1$
SET	6,(IX+zz)	DD CB zz F6	23	$(<IX>+zz)_6 <- 1$
SET	6,(IY+zz)	FD CB zz F6	23	$(<IY>+zz)_6 <- 1$
SET	6,A	CB F7	8	$A_6 <- 1$
SET	6,B	CB F0	8	$B_6 <- 1$
SET	6,C	CB F1	8	$C_6 <- 1$
SET	6,D	CB F2	8	$D_6 <- 1$
SET	6,E	CB F3	8	$E_6 <- 1$
SET	6,H	CB F4	8	$H_6 <- 1$
SET	6,L	CB F5	8	$L_6 <- 1$
SET	7,(HL)	CB FE	15	$(<HL>)_7 <- 1$
SET	7,(IX+zz)	DD CB zz FE	23	$(<IX>+zz)_7 <- 1$
SET	7,(IY+zz)	FD CB zz FE	23	$(<IY>+zz)_7 <- 1$
SET	7,A	CB FF	8	$A_7 <- 1$
SET	7,B	CB F8	8	$B_7 <- 1$
SET	7,C	CB F9	8	$C_7 <- 1$
SET	7,D	CB FA	8	$D_7 <- 1$

SET	7,E	CB FB	8	$E_7 \leftarrow 1$
SET	7,H	CB FC	8	$H_7 \leftarrow 1$
SET	7,L	CB FD	8	$L_7 \leftarrow 1$
SLA	(HL)	CB 26	15	$CY \leftarrow <(<HL>)>_7$ $(<HL>)_{7,\ldots,1} \leftarrow <(<HL>)>_{6,\ldots,0}$ $(<HL>)_0 \leftarrow 0$
SLA	(IX+zz)	DD CB zz 26	23	$CY \leftarrow <(<IX>+zz)>_7$ $(<IX>+zz)_{7,\ldots,1} \leftarrow <(<IX>+zz)>_{6,\ldots,0}$ $(<IX>+zz)_0 \leftarrow 0$
SLA	(IY+zz)	FD CB zz 26	23	$CY \leftarrow <(<IY>+zz)>_7$ $(<IY>+zz)_{7,\ldots,1} \leftarrow <(<IY>+zz)>_{6,\ldots,0}$ $(<IY>+zz)_0 \leftarrow 0$
SLA	A	CB 27	8	$CY \leftarrow <A>_7$ $A_{7,\ldots,1} \leftarrow <A>_{6,\ldots,0}$ $A_0 \leftarrow 0$
SLA	B	CB 20	8	$CY \leftarrow <B>_7$ $B_{7,\ldots,1} \leftarrow <B>_{6,\ldots,0}$ $B_0 \leftarrow 0$
SLA	C	CB 21	8	$CY \leftarrow <C>_7$ $C_{7,\ldots,1} \leftarrow <C>_{6,\ldots,0}$ $C_0 \leftarrow 0$
SLA	D	CB 22	8	$CY \leftarrow <D>_7$ $D_{7,\ldots,1} \leftarrow <D>_{6,\ldots,0}$ $D_0 \leftarrow 0$
SLA	E	CB 23	8	$CY \leftarrow <E>_7$ $E_{7,\ldots,1} \leftarrow <E>_{6,\ldots,0}$ $E_0 \leftarrow 0$
SLA	H	CB 24	8	$CY \leftarrow <H>_7$ $H_{7,\ldots,1} \leftarrow <H>_{6,\ldots,0}$ $H_0 \leftarrow 0$
SLA	L	CB 25	8	$CY \leftarrow <L>_7$ $L_{7,\ldots,1} \leftarrow <L>_{6,\ldots,0}$ $L_0 \leftarrow 0$
SRA	(HL)	CB 2E	15	$CY \leftarrow <(<HL>)>_0$ $(<HL>)_{0,\ldots,6} \leftarrow <(<HL>)>_{1,\ldots,7}$
SRA	(IX+zz)	DD CB zz 2E	23	$CY \leftarrow <(<IX>+zz)>_0$ $(<IX>+zz)_{0,\ldots,6} \leftarrow <(<IX>+zz)>_{1,\ldots,7}$
SRA	(IY+zz)	FD CB zz 2E	23	$CY \leftarrow <(<IY>+zz)>_0$ $(<IY>+zz)_{0,\ldots,6} \leftarrow <(<IY>+zz)>_{1,\ldots,7}$
SRA	A	CB 2F	8	$CY \leftarrow <A>_0$ $A_{0,\ldots,6} \leftarrow <A>_{1,\ldots,7}$

SRA	B	CB 28	8	CY <– <B>$_0$ <B>$_{0,...,6}$ <– <B>$_{1,...,7}$
SRA	C	CB 29	8	CY <– <C>$_0$ <C>$_{0,...,6}$ <– <C>$_{1,...,7}$
SRA	D	CB 2A	8	CY <– <D>$_0$ <D>$_{0,...,6}$ <– <D>$_{1,...,7}$
SRA	E	CB 2B	8	CY <– <E>$_0$ <E>$_{0,...,6}$ <– <E>$_{1,...,7}$
SRA	H	CB 2C	8	CY <– <H>$_0$ <H>$_{0,...,6}$ <– <H>$_{1,...,7}$
SRA	L	CB 2D	8	CY <– <L>$_0$ <L>$_{0,...,6}$ <– <L>$_{1,...,7}$
SRL	(HL)	CB 3E	15	CY <– <(<HL>)>$_0$ <HL>)$_{0,...,6}$ <– <(<HL>)>$_{1,...,7}$ (<HL>)$_7$ <– 0
SRL	(IX+zz)	DD CB zz 3E	23	CY <– <(<IX>+zz)>$_0$ (<IX>+zz)$_{0,...,6}$ <– <(<IX>+zz)>$_{1,...,7}$ (<IX>+zz)$_7$ <– 0
SRL	(IY+zz)	FD CB zz 3E	23	CY <– <(<IY>+zz)>$_0$ (<IY>+zz)$_{0,...,6}$ <– <(<IY>+zz)>$_{1,...,7}$ (<IY>+zz)$_7$ <– 0
S	A	CB 3F	8	CY <– <A>$_0$ <A>$_{0,...,6}$ <– <A>$_{1,...,7}$ A$_7$ <– 0
SRL	B	CB 38	8	CY <– <B>$_0$ <B>$_{0,...,6}$ <– <B>$_{1,...,7}$ B$_7$ <– 0
SRL	C	CB 39	8	CY <– <C>$_0$ <C>$_{0,...,6}$ <– <C>$_{1,...,7}$ C$_7$ <– 0
SRL	D	CB 3A	8	CY <– <D>$_0$ <D>$_{0,...,6}$ <– <D>$_{1,...,7}$ D$_7$ <– 0
SRL	E	CB 3B	8	CY <– <E>$_0$ <E>$_{0,...,6}$ <– <E>$_{1,...,7}$ E$_7$ <– 0
SRL	H	CB 3C	8	CY <– <H>$_0$ <H>$_{0,...,6}$ <– <H>$_{1,...,7}$ H$_7$ <– 0
SRL	L	CB 3D	8	CY <– <L>$_0$ <L>$_{0,...,6}$ <– <L>$_{1,...,7}$ L$_7$ <– 0

SUB	(HL)	96	7	A<–<A>–<(<HL>)>
SUB	(IX+zz)	DD 96 zz	19	A<–<A>–<(<IX>+zz)>
SUB	(IY+zz)	FD 96 zz	19	A<–<A>–<(<IY>+zz)>
SUB	A	97	4	A<–<A>–<A>
SUB	B	90	4	A<–<A>–<B>
SUB	C	91	4	A<–<A>–<C>
SUB	D	92	4	A<–<A>–<D>
SUB	E	93	4	A<–<A>–<E>
SUB	H	94	4	A<–<A>–<H>
SUB	L	95	4	A<–<A>–<L>
SUB	xx	D6 xx	7	A<–<A>–xx
XOR	(HL)	AE	7	A<–<A> xor <(<HL>)>
XOR	(IX+zz)	DD AE zz	19	A<–<A> xor <(<IX>+zz)>
XOR	(IY+zz)	FD AE zz	19	A<–<A> xor <(<IY>+zz)>
XOR	A	AF	4	A<–<A> xor <A>
XOR	B	A8	4	A<–<A> xor <B>
XOR	C	A9	4	A<–<A> xor <C>
XOR	D	AA	4	A<–<A> xor <D>
XOR	E	AB	4	A<–<A> xor <E>
XOR	H	AC	4	A<–<A> xor <H>
XOR	L	AD	4	A<–<A> xor <L>
XOR	xx	EE xx	7	A<–<A> xor xx

Anhang C

Verzeichnis der Assembler-Befehle (sortiert nach Objekt-Codes)

Im nachfolgenden Verzeichnis sind in der ersten beziehungsweise dritten Spalte die Objekt-Codes aufgelistet, in der zweiten beziehungsweise vierten Spalte die zugehörige Interpretation als Assembler-Befehl. Die Objekt-Codes haben eine Länge von 1 bis 4 Bytes. Es gelten ansonsten die Konventionen aus Anhang B.

00	NOP	30	JR NC,tt
01 yy xx	LD BC,xxyy	31 yy xx	LD SP,xxyy
02	LD (BC),A	32 yy xx	LD (xxyy),A
03	INC BC	33	INC SP
04	INC B	34	INC (HL)
05	DEC B	35	DEC (HL)
06 xx	LD B,xx	36 xx	LD (HL),xx
07	RLCA	37	SCF
08	EX AF,AF'	38 tt	JR C,tt
09	ADD HL,BC	39	ADD HL,SP
0A	LD A,(BC)	3A yy xx	LD A,(xxyy)
0B	DEC BC	3B	DEC SP
0C	INC C	3C	INC A
0D	DEC C	3D	DEC A
0E xx	LD C,xx	3E xx	LD A,xx
0F	RRCA	3F	CCF
10 tt	DJNZ tt	40	LD B,B
11 yy xx	LD DE,xxyy	41	LD B,C
12	LD (DE),A	42	LD B,D
13	INC DE	43	LD B,E
14	INC D	44	LD B,H
15	DEC D	45	LD B,L

16	xx	LD	D,xx	46	LD	B,(HL)
17		RLA		47	LD	B,A
18	tt	JR	tt	48	LD	C,B
19		ADD	HL,DE	49	LD	C,C
1A		LD	A,(DE)	4A	LD	C,D
1B		DEC	DE	4B	LD	C,E
1C		INC	E	4C	LD	C,H
1D		DEC	E	4D	LD	C,L
1E	xx	LD	E,xx	4E	LD	C,(HL)
1F		RRA		4F	LD	C,A
20	tt	JR	NZ,tt	50	LD	D,B
21	yy xx	LD	HL,xxyy	51	LD	D,C
22	yy xx	LD	(xxyy),HL	52	LD	D,D
23		INC	HL	53	LD	D,E
24		INC	H	54	LD	D,H
25		DEC	H	55	LD	D,L
26	xx	LD	H,xx	56	LD	D,(HL)
27		DAA		57	LD	D,A
28	tt	JR	Z,tt	58	LD	E,B
29		ADD	HL,HL	59	LD	E,C
2A	yy xx	LD	HL,(xxyy)	5A	LD	E,D
2B		DEC	HL	5B	LD	E,E
2C		INC	L	5E	LD	E,H
2D		DEC	L	5D	LD	E,L
2E	xx	LD	L,xx	5E	LD	E,(HL)
2F		CPL		5F	LD	E,A

60	LD	H,B	9B	SBC	A,E
61	LD	H,C	9E	SBC	A,H
62	LD	H,D	9D	SBC	A,L
63	LD	H,E	9E	SBC	A,(HL)
64	LD	H,H	9F	SBC	A,A
65	LD	H,L	A0	AND	B
66	LD	H,(HL)	A1	AND	C
67	LD	H,A	A2	AND	D
68	LD	L,B	A3	AND	E
69	LD	L,C	A4	AND	H
6A	LD	L,D	A5	AND	L
6B	LD	L,E	A6	AND	(HL)
6E	LD	L,H	A7	AND	A
6D	LD	L,L	A8	XOR	B
6E	LD	L,(HL)	A9	XOR	C
6F	LD	L,A	AA	XOR	D

70	LD	(HL),B	AB		XOR	E
71	LD	(HL),C	AE		XOR	H
72	LD	(HL),D	AD		XOR	L
73	LD	(HL),E	AE		XOR	(HL)
74	LD	(HL),H	AF		XOR	A
75	LD	(HL),L	B0		OR	B
76	HALT		B1		OR	C
77	LD	(HL),A	B2		OR	D
78	LD	A,B	B3		OR	E
79	LD	A,C	B4		OR	H
7A	LD	A,D	B5		OR	L
7B	LD	A,E	B6		OR	(HL)
7E	LD	A,H	B7		OR	A
7D	LD	A,L	B8		CP	B
7E	LD	A,(HL)	B9		CP	C
7F	LD	A,A	BA		CP	D
80	ADD	A,B	BB		CP	E
81	ADD	A,C	BE		CP	H
82	ADD	A,D	BD		CP	L
83	ADD	A,E	BE		CP	(HL)
84	ADD	A,H	BF		CP	A
85	ADD	A,L	C0		RET	NZ
86	ADD	A,(HL)	C1		POP	BC
87	ADD	A,A	C2	yy xx	JP	NZ,xxyy
88	ADC	A,B	C3	yy xx	JP	xxyy
89	ADC	A,C	C4	yy xx	CALL	NZ,xxyy
8A	ADC	A,D	C5		PUSH	BC
8B	ADC	A,E	C6	xx	ADD	A,xx
8E	ADC	A,H	C7		RST	00H
8D	ADC	A,L	C8		RET	Z
8E	ADC	A,(HL)	C9		RET	
8F	ADC	A,A	CA	yy xx	JP	Z,xxyy
90	SUB	B	CB	00	RLC	B
91	SUB	C	CB	01	RLC	C
92	SUB	D	CB	02	RLC	D
93	SUB	E	CB	03	RLC	E
94	SUB	H	CB	04	RLC	H
95	SUB	L	CB	05	RLC	L
96	SUB	(HL)	CB	06	RLC	(HL)
97	SUB	A	CB	07	RLC	A
98	SBC	A,B	CB	08	RRC	B
99	SBC	A,C	CB	09	RRC	C
9A	SBC	A,D	CB	0A	RRC	D

CB 0B	RRC E	CB 4E	BIT 1,(HL)
CB 0C	RRC H	CB 4F	BIT 1,A
CB 0D	RRC L	CB 50	BIT 2,B
CB 0E	RRC (HL)	CB 51	BIT 2,C
CB 0F	RRC A	CB 52	BIT 2,D
CB 10	RL B	CB 53	BIT 2,E
CB 11	RL C	CB 54	BIT 2,H
CB 12	RL D	CB 55	BIT 2,L
CB 13	RL E	CB 56	BIT 2,(HL)
CB 14	RL H	CB 57	BIT 2,A
CB 15	RL L	CB 58	BIT 3,B
CB 16	RL (HL)	CB 59	BIT 3,C
CB 17	RL A	CB 5A	BIT 3,D
CB 18	RR B	CB 5B	BIT 3,E
CB 19	RR C	CB 5C	BIT 3,H
CB 1A	RR D	CB 5D	BIT 3,L
CB 1B	RR E	CB 5E	BIT 3,(HL)
CB 1C	RR H	CB 5F	BIT 3,A
CB 1D	RR L	CB 60	BIT 4,B
CB 1E	RR (HL)	CB 61	BIT 4,C
CB 1F	RR A	CB 62	BIT 4,D
CB 20	SLA B	CB 63	BIT 4,E
CB 21	SLA C	CB 64	BIT 4,H
CB 22	SLA D	CB 65	BIT 4,L
CB 23	SLA E	CB 66	BIT 4,(HL)
CB 24	SLA H	CB 67	BIT 4,A
CB 25	SLA L	CB 68	BIT 5,B
CB 26	SLA (HL)	CB 69	BIT 5,C
CB 27	SLA A	CB 6A	BIT 5,D
CB 28	SRA B	CB 6B	BIT 5,E
CB 29	SRA C	CB 6C	BIT 5,H
CB 2A	SRA D	CB 6D	BIT 5,L
CB 2B	SRA E	CB 6E	BIT 5,(HL)
CB 2C	SRA H	CB 6F	BIT 5,A
CB 2D	SRA L	CB 70	BIT 6,B
CB 2E	SRA (HL)	CB 71	BIT 6,C
CB 2F	SRA A	CB 72	BIT 6,D
CB 38	SRL B	CB 73	BIT 6,E
CB 39	SRL C	CB 74	BIT 6,H
CB 3A	SRL D	CB 75	BIT 6,L
CB 3B	SRL E	CB 76	BIT 6,(HL)
CB 3C	SRL H	CB 77	BIT 6,A
CB 3D	SRL L	CB 78	BIT 7,B

CB	3E	SRL	(HL)	CB	79	BIT	7,C
CB	3F	SRL	A	CB	7A	BIT	7,D
CB	40	BIT	0,B	CB	7B	BIT	7,E
CB	41	BIT	0,C	CB	7C	BIT	7,H
CB	42	BIT	0,D	CB	7D	BIT	7,L
CB	43	BIT	0,E	CB	7E	BIT	7,(HL)
CB	44	BIT	0,H	CB	7F	BIT	7,A
CB	45	BIT	0,L	CB	80	RES	0,B
CB	46	BIT	0,(HL)	CB	81	RES	0,C
CB	47	BIT	0,A	CB	82	RES	0,D
CB	48	BIT	1,B	CB	83	RES	0,E
CB	49	BIT	1,C	CB	84	RES	0,H
CB	4A	BIT	1,D	CB	85	RES	0,L
CB	4B	BIT	1,E	CB	86	RES	0,(HL)
CB	4C	BIT	1,H	CB	87	RES	0,A
CB	4D	BIT	1,L	CB	88	RES	1,B
CB	89	RES	1,C	CB	C4	SET	0,H
CB	8A	RES	1,D	CB	C5	SET	0,L
CB	8B	RES	1,E	CB	C6	SET	0,(HL)
CB	8C	RES	1,H	CB	C7	SET	0,A
CB	8D	RES	1,L	CB	C8	SET	1,B
CB	8E	RES	1,(HL)	CB	C9	SET	1,C
CB	8F	RES	1,A	CB	CA	SET	1,D
CB	90	RES	2,B	CB	CB	SET	1,E
CB	91	RES	2,C	CB	CC	SET	1,H
CB	92	RES	2,D	CB	CD	SET	1,L
CB	93	RES	2,E	CB	CE	SET	1,(HL)
CB	94	RES	2,H	CB	CF	SET	1,A
CB	95	RES	2,L	CB	D0	SET	2,B
CB	96	RES	2,(HL)	CB	D1	SET	2,C
CB	97	RES	2,A	CB	D2	SET	2,D
CB	98	RES	3,B	CB	D3	SET	2,E
CB	99	RES	3,C	CB	D4	SET	2,H
CB	9A	RES	3,D	CB	D5	SET	2,L
CB	9B	RES	3,E	CB	D6	SET	2,(HL)
CB	9C	RES	3,H	CB	D7	SET	2,A
CB	9D	RES	3,L	CB	D8	SET	3,B
CB	9E	RES	3,(HL)	CB	D9	SET	3,C
CB	9F	RES	3,A	CB	DA	SET	3,D
CB	A0	RES	4,B	CB	DB	SET	3,E
CB	A1	RES	4,C	CB	DC	SET	3,H
CB	A2	RES	4,D	CB	DD	SET	3,L

```
CB A3       RES  4,E        CB DE       SET  3,(HL)
CB A4       RES  4,H        CB DF       SET  3,A
CB A5       RES  4,L        CB E0       SET  4,B
CB A6       RES  4,(HL)     CB E1       SET  4,C
CB A7       RES  4,A        CB E2       SET  4,D
CB A8       RES  5,B        CB E3       SET  4,E
CB A9       RES  5,C        CB E4       SET  4,H
CB AA       RES  5,D        CB E5       SET  4,L
CB AB       RES  5,E        CB E6       SET  4,(HL)
CB AC       RES  5,H        CB E7       SET  4,A
CB AD       RES  5,L        CB E8       SET  5,B
CB AE       RES  5,(HL)     CB E9       SET  5,C
CB AF       RES  5,A        CB EA       SET  5,D
CB B0       RES  6,B        CB EB       SET  5,E
CB B1       RES  6,C        CB EC       SET  5,H
CB B2       RES  6,D        CB ED       SET  5,L
CB B3       RES  6,E        CB EE       SET  5,(HL)
CB B4       RES  6,H        CB EF       SET  5,A
CB B5       RES  6,L        CB F0       SET  6,B
CB B6       RES  6,(HL)     CB F1       SET  6,C
CB B7       RES  6,A        CB F2       SET  6,D
CB B8       RES  7,B        CB F3       SET  6,E
CB B9       RES  7,C        CB F4       SET  6,H
CB BA       RES  7,D        CB F5       SET  6,L
CB BB       RES  7,E        CB F6       SET  6,(HL)
CB BC       RES  7,H        CB F7       SET  6,A
CB BD       RES  7,L        CB F8       SET  7,B
CB BE       RES  7,(HL)     CB F9       SET  7,C
CB BF       RES  7,A        CB FA       SET  7,D
CB C0       SET  0,B        CB FB       SET  7,E
CB C1       SET  0,C        CB FC       SET  7,H
CB C2       SET  0,D        CB FD       SET  7,L
CB C3       SET  0,E        CB FE       SET  7,(HL)

CB FF       SET  7,A        DD CB zz 46 BIT  0,(IX+zz)
CC yy xx    CALL Z,xxyy     DD CB zz 4E BIT  1,(IX+zz)
CD yy xx    CALL xxyy       DD CB zz 56 BIT  2,(IX+zz)
CE xx       ADC  A,xx       DD CB zz 5E BIT  3,(IX+zz)
CF          RST  08H        DD CB zz 66 BIT  4,(IX+zz)
D0          RET  NC         DD CB zz 6E BIT  5,(IX+zz)
D1          POP  DE         DD CB zz 76 BIT  6,(IX+zz)
D2 yy xx    JP   NC,xxyy    DD CB zz 7E BIT  7,(IX+zz)
D3 xx       OUT  (xx),A     DD CB zz 86 RES  0,(IX+zz)
```

```
D4 yy xx    CALL NC,xxyy
D5          PUSH DE
D6 xx       SUB  xx
D7          RST  10H
D8          RET  C
D9          EXX
DA yy xx    JP   C,xxyy
DB xx       IN   A,(xx)
DC yy xx    CALL C,xxyy
DD 09       ADD  IX,BC
DD 19       ADD  IX,DE
DD 21 yy xx LD   IX,xxyy
DD 22 yy xx LD   (xxyy),IX
DD 23       INC  IX
DD 29       ADD  IX,IX
DD 2A yy xx LD   IX,(xxyy)
DD 2B       DEC  IX
DD 34 zz    INC  (IX+zz)
DD 35 zz    DEC  (IX+zz)
DD 36 zz xx LD   (IX+zz),xx
DD 39       ADD  IX,SP
DD 46 zz    LD   B,(IX+zz)
DD 4E zz    LD   C,(IX+zz)
DD 56 zz    LD   D,(IX+zz)
DD 5E zz    LD   E,(IX+zz)
DD 66 zz    LD   H,(IX+zz)
DD 6E zz    LD   L,(IX+zz)
DD 70 zz    LD   (IX+zz),B
DD 71 zz    LD   (IX+zz),C
DD 72 zz    LD   (IX+zz),D
DD 73 zz    LD   (IX+zz),E
DD 74 zz    LD   (IX+zz),H
DD 75 zz    LD   (IX+zz),L
DD 77 zz    LD   (IX+zz),A
DD 7E zz    LD   A,(IX+zz)
DD 86 zz    ADD  A,(IX+zz)
DD 8E zz    ADC  A,(IX+zz)
DD 96 zz    SUB  (IX+zz)
DD 9E zz    SBC  A,(IX+zz)
DD A6 zz    AND  (IX+zz)
DD AE zz    XOR  (IX+zz)
DD B6 zz    OR   (IX+zz)
DD BE zz    CP   (IX+zz)
DD CB zz 8E RES  1,(IX+zz)
DD CB zz 96 RES  2,(IX+zz)
DD CB zz 9E RES  3,(IX+zz)
DD CB zz A6 RES  4,(IX+zz)
DD CB zz AE RES  5,(IX+zz)
DD CB zz B6 RES  6,(IX+zz)
DD CB zz BE RES  7,(IX+zz)
DD CB zz C6 SET  0,(IX+zz)
DD CB zz CE SET  1,(IX+zz)
DD CB zz D6 SET  2,(IX+zz)
DD CB zz DE SET  3,(IX+zz)
DD CB zz E6 SET  4,(IX+zz)
DD CB zz EE SET  5,(IX+zz)
DD CB zz F6 SET  6,(IX+zz)
DD CB zz FE SET  7,(IX+zz)
DD E1       POP  IX
DD E3       EX   (SP),IX
DD E5       PUSH IX
DD E9       JP   (IX)
DD F9       LD   SP,IX
DE xx       SBC  A,xx
DF          RST  18H
E0          RET  PO
E1          POP  HL
E2 yy xx    JP   PO,xxyy
E3          EX   (SP),HL
E4 yy xx    CALL PO,xxyy
E5          PUSH HL
E6 xx       AND  xx
E7          RST  20H
E8          RET  PE
E9          JP   (HL)
EA yy xx    JP   PE,xxyy
EB          EX   DE,HL
EC yy xx    CALL PE,xxyy
ED 40       IN   B,(C)
ED 41       OUT  (C),B
ED 42       SBC  HL,BC
ED 43 yy xx LD   (xxyy),BC
ED 44       NEG
ED 45       RETN
ED 46       IM   0
ED 47       LD   I,A
```

```
DD CB zz 06    RLC   (IX+zz)
DD CB zz 0E    RRC   (IX+zz)
DD CB zz 16    RL    (IX+zz)
DD CB zz 1E    RR    (IX+zz)
DD CB zz 26    SLA   (IX+zz)
DD CB zz 2E    SRA   (IX+zz)
DD CB zz 3E    SRL   (IX+zz)

ED 51          OUT   (C),D
ED 52          SBC   HL,DE
ED 53 yy xx    LD    (xxyy),DE
ED 56          IM    1
ED 57          LD    A,I
ED 58          IN    E,(C)
ED 59          OUT   (C),E
ED 5A          ADC   HL,DE
ED 5B yy xx    LD    DE,(xxyy)
ED 5E          IM    2
ED 5F          LD    A,R
ED 60          IN    H,(C)
ED 61          OUT   (C),H
ED 62          SBC   HL,HL
ED 63          LD    (xxyy),HL
ED 67          RRD
ED 68          IN    L,(C)
ED 69          OUT   (C),L
ED 6A          ADC   HL,HL
ED 6B          LD    HL,(xxyy)
ED 6F          RLD
ED 72          SBC   HL,SP
ED 73 yy xx    LD    (xxyy),SP
ED 78          IN    A,(C)
ED 79          OUT   (C),A
ED 7A          ADC   HL,SP
ED 7B yy xx    LD    SP,(xxyy)
ED A0          LDI
ED A1          CPI
ED A2          INI
ED A3          OUTI
ED A8          LDD
ED A9          CPD
ED AA          IND
ED AB          OUTD

ED 48          IN    C,(C)
ED 49          OUT   (C),C
ED 4A          ADC   HL,BC
ED 4B          LD    BC,(xxyy)
ED 4D          RETI
ED 4F          LD    R,A
ED 50          IN    D,(C)

FD 19          ADD   IY,DE
FD 21 yy xx    LD    IY,xxyy
FD 22 yy xx    LD    (xxyy),IY
FD 23          INC   IY
FD 29          ADD   IY,IY
FD 2A yy xx    LD    IY,(xxyy)
FD 2B          DEC   IY
FD 34 zz       INC   (IY+zz)
FD 35 zz       DEC   (IY+zz)
FD 36 zz xx    LD    (IY+zz),xx
FD 39          ADD   IY,SP
FD 46 zz       LD    B,(IY+zz)
FD 4E zz       LD    C,(IY+zz)
FD 56 zz       LD    D,(IY+zz)
FD 5E zz       LD    E,(IY+zz)
FD 66 zz       LD    H,(IY+zz)
FD 6E zz       LD    L,(IY+zz)
FD 70 zz       LD    (IY+zz),B
FD 71 zz       LD    (IY+zz),C
FD 72 zz       LD    (IY+zz),D
FD 73 zz       LD    (IY+zz),E
FD 74 zz       LD    (IY+zz),H
FD 75 zz       LD    (IY+zz),L
FD 77 zz       LD    (IY+zz),A
FD 7E zz       LD    A,(IY+zz)
FD 86 zz       ADD   A,(IY+zz)
FD 8E zz       ADC   A,(IY+zz)
FD 96 zz       SUB   (IY+zz)
FD 9E zz       SBC   A,(IY+zz)
FD A6 zz       AND   (IY+zz)
FD AE zz       XOR   (IY+zz)
FD B6 zz       OR    (IY+zz)
FD BE zz       CP    (IY+zz)
FD CB zz 06    RLC   (IY+zz)
FD CB zz 0E    RRC   (IY+zz)
```

```
ED B0          LDIR
ED B1          CPIR
ED B2          INIR
ED B3          OTIR
ED B8          LDDR
ED B9          CPDR
ED BA          INDR
ED BB          OTDR
EE xx          XOR   xx
EF             RST   28H
F0             RET   P
F1             POP   AF
F2 yy xx       JP    P,xxyy
F3             DI
F4 yy xx       CALL  P,xxyy
F5             PUSH  AF
F6 xx          OR    xx
F7             RST   30H
F8             RET   M
F9             LD    SP,HL
FA yy xx       JP    M,xxyy
FB             EI
FC yy xx       CALL  M,xxyy
FD 09          ADD   IY,BC

FD CB zz DE    SET   3,(IY+zz)
FD CB zz E6    SET   4,(IY+zz)
FD CB zz EE    SET   5,(IY+zz)
FD CB zz F6    SET   6,(IY+zz)
FD CB zz FE    SET   7,(IY+zz)
FD E1          POP   IY
FD E3          EX    (SP),IY
FD E5          PUSH  IY
FD E9          JP    (IY)
FD F9          LD    SP,IY
FE xx          CP    xx
FF             RST   38H
```

```
FD CB zz 16    RL    (IY+zz)
FD CB zz 1E    RR    (IY+zz)
FD CB zz 26    SLA   (IY+zz)
FD CB zz 2E    SRA   (IY+zz)
FD CB zz 3E    SRL   (IY+zz)
FD CB zz 46    BIT   0,(IY+zz)
FD CB zz 4E    BIT   1,(IY+zz)
FD CB zz 56    BIT   2,(IY+zz)
FD CB zz 5E    BIT   3,(IY+zz)
FD CB zz 66    BIT   4,(IY+zz)
FD CB zz 6E    BIT   5,(IY+zz)
FD CB zz 76    BIT   6,(IY+zz)
FD CB zz 7E    BIT   7,(IY+zz)
FD CB zz 86    RES   0,(IY+zz)
FD CB zz 8E    RES   1,(IY+zz)
FD CB zz 96    RES   2,(IY+zz)
FD CB zz 9E    RES   3,(IY+zz)
FD CB zz A6    RES   4,(IY+zz)
FD CB zz AE    RES   5,(IY+zz)
FD CB zz B6    RES   6,(IY+zz)
FD CB zz BE    RES   7,(IY+zz)
FD CB zz C6    SET   0,(IY+zz)
FD CB zz CE    SET   1,(IY+zz)
FD CB zz D6    SET   2,(IY+zz)
```

Anhang D

Systematischer Aufbau des Befehlssatzes

Die folgende Darstellung soll zum tieferen Verständnis des Befehlssatzes sowie für den Entwurf eines Disassemblers dienen.

Ein Teil der Befehle des Z80 wird durch das erste Byte des Objekt-Codes determiniert; es können dann noch ein oder zwei Bytes für direkte Operanden und Adressen folgen.

Vier bestimmte Werte fungieren als »Escape-Codes«: CB, DD, ED, FD. Mit CB (= 11001011 binär) wird ein Rotations- oder Verschiebe-Befehl beziehungsweise ein Bit-Manipulations-Befehl eingeleitet. Mit E (= 11101101 binär) beginnende Befehle realisieren eine Klasse von recht unterschiedlichen Spezialoperationen.

Mit DD beziehungsweise FD werden Befehle eingeleitet, die sich auf das Indexregister IX beziehungsweise IY beziehen. Der darauf folgende Rest des Objekt-Codes kann bis zu drei Bytes umfassen und enthält meist eine Relativadresse; durch Weglassen des Codes DD (beziehungsweise FD) und der eventuell vorhandenen Relativadresse ergibt sich ein Objekt-Code, der sich als Objekt-Code eines entsprechenden Befehls bezüglich des HL-Registers interpretieren läßt. Die Relativadresse – falls vorhanden – belegt stets das dritte Byte in der Folge.

In der Zusammenstellung steht je ein Byte des Objekt-Codes in einer eigenen Zeile. Die Angabe erfolgt bitweise (numerisch beziehungsweise symbolisch), byteweise (symbolisch xx oder tt) und wortweise (symbolisch xxyy), wobei die Symbole für einzelne Bits oder kleine Gruppen von Bits je nach Kontext ihre Bedeutung ändern und deswegen an der Stelle ihrer Verwendung erklärt werden.

Die Bemerkung »Indexregister + Relativadresse« bedeutet, daß es den auf das HL-Register bezogenen Befehl auch für die Indexregister gibt, wobei eine Relativadresse verwendet wird. Fehlt der Zusatz »+ Relativadresse«, so wird keine Relativadresse verwendet.

Wir behandeln zuerst alle Befehle, deren Objekt-Code mit 00 beginnt:

00 r 110 (Indexregister + Relativadresse)
xx

8-Bit-Lade-Befehle der Form LD r,xx
dabei bedeutet r = 000 B
001 C
010 D
011 E
100 H
101 L
110 (HL)
111 A

0011 f 010
yy
xx

8-Bit-Lade-Befehle auf dem A-Register
dabei bedeutet f = 0 LD (xxyy),A
1 LD A,(xxyy)

000 f 010

8-Bit-Lade-Befehle auf dem A-Register
dabei bedeutet f = 00 LD (BC),A
01 LD A,(BC)
10 LD (DE),A
11 LD A,(DE)

00 r 10 f (Indexregister + Relativadresse)

8-Bit-Arithmetik-Befehle der Form INC r oder DEC r
dabei bedeutet r = 000 B f = 0 INC
001 C 1 DEC
010 D
011 E
100 H
101 L
110 (HL)
111 A

00 dd 0001 (Indexregister)
yy
xx

16-Bit-Lade-Befehle der Form LD dd,xxyy
dabei bedeutet dd = 00 BC
01 DE
10 HL
11 SP

0010 f 010 (Indexregister)
yy
xx

16-Bit-Lade-Befehle auf dem HL-Register
dabei bedeutet f = 0 LD (xxyy),HL
1 LD HL,(xxyy)

00 dd 1001 (Indexregister)

16-Bit-Arithmetik-Befehle der Form ADD HL,dd
dabei bedeutet dd = 00 BC
01 DE
10 HL

00 dd f 011 (Indexregister)

16-Bit-Arithmetik-Befehle der Form INC dd oder DEC dd
dabei bedeutet dd = 00 BC
01 DE
10 HL
11 SP
f = 0 INC
1 DEC

000 f 111

Rotations-Befehle auf dem A-Register
dabei bedeutet f = 00 RLCA
01 RRCA
10 RLA
11 RRA

0001 f 000
tt

Relative Sprünge
dabei bedeutet f = 0 DJNZ tt
1 JR tt

001 cc 000
tt

Relative Sprünge der Form JR cc,tt
dabei bedeutet cc = 00 NZ
01 Z
10 NC
11 C

0011 f 111

Befehle zur Beeinflussung des Carry-Flags
dabei bedeutet f = 0 SCF
1 CCF

Weitere Befehle, deren Objekt-Code mit 00 beginnt, sind:

00000000	NOP
00001000	EX AF,AF'
00100111	DAA
00101111	CPL

Nun kommen die Befehle an die Reihe, deren Objekt-Code mit 01 beginnt:

01 r r' (Indexregister + Relativadresse)

8-Bit-Lade-Befehle der Form LD r,r'

dabei bedeutet	r =		r' =	
	000	B	000	B
	001	C	001	C
	010	D	010	D
	011	E	011	E
	100	H	100	H
	101	L	101	L
	110	(HL)	110	(HL)
	111	A	111	A

Die Kombination r = 110 und r' = 110 kommt nicht vor. Sie steht für den Befehl HALT:

01110110 HALT

Es folgen die Befehle, deren Objekt-Code mit 10 beginnt:

10 f r (Indexregister + Relativadresse)

Arithmetik/Logik-Befehle der Form f r beziehungsweise f A,r
dabei bedeutet

f =	000	ADD	
	001	ADC	
	010	SUB	
	011	SBC	
	100	AND	
	101	XOR	
	110	OR	
	111	CP	

r =	000	B
	001	C
	010	D
	011	E
	100	H
	101	L
	110	(HL)
	111	A

Der Objekt-Code aller übrigen Befehle beginnt mit 11. Wir listen diese auf mit Ausnahme der Befehle, deren Objekt-Code mit DD oder FD beginnt. Die Befehle, deren Objekt-Code mit CB oder ED beginnt, stellen wir dabei an das Ende:

11 f 110 (Indexregister + Relativadresse)
xx

Arithmetik/Logik-Befehle der Form f xx beziehungsweise f A,xx
dabei bedeutet

f =	000	ADD
	001	ADC
	010	SUB
	011	SBC
	100	AND
	101	XOR
	110	OR
	111	CP

11 dd 0 f 01 (Indexregister)

16-Bit-Lade-Befehle der Form PUSH dd oder POP dd
dabei bedeutet

dd =	00	BC
	01	DE
	10	HL
	11	AF

f =	0	POP
	1	PUSH

11111001 (Indexregister)

16-Bit-Lade-Befehl LD SP,HL

11100011 (Indexregister)

Austausch-Befehl EX (SP),HL

1101 f 011
xx

Ein-/Ausgabe-Befehle mit direkter Portadresse
dabei bedeutet f = 0 OUT (xx),A
1 IN A,(xx)

11 cc 000

Bedingte Unterprogramm-Rücksprung-Befehle der Form RET cc
dabei bedeutet cc = 000 NZ
001 Z
010 NC
011 C
100 PO
101 PE
110 P
111 M

11 cc 010
yy
xx

Absolute bedingte Sprung-Befehle der Form JP cc,xxyy
dabei bedeutet cc = 000 NZ
001 Z
010 NC
011 C
100 PO
101 PE
110 P
111 M

11 cc 100
yy
xx

Bedingte Unterprogramm-Aufruf-Befehle der Form CALL cc,xxyy
dabei bedeutet cc = 000 NZ
001 Z
010 NC
011 C
100 PO
101 PE
110 P
111 M

11101001 (Indexregister)

Indirekter Sprung-Befehl JP (HL)

11 p 111

Restart-Befehle der Form RST p
dabei bedeutet t = 000 00H
001 08H
010 10H
011 18H
100 20H
101 28H
110 30H
111 38H

1111 f 011

Unterbrechungs-Steuerungs-Befehle
dabei bedeutet f = 0 DI
1 EI

Weitere Befehle, deren Objekt-Code mit 11 beginnt, sind:

11001001	RET
11000011	JP xxyy
yy	
xx	
11001101	CALL xxyy
yy xx	
11011001	EXX
11101011	EX DE,HL

Es folgen die Befehle, deren Objekt-Code mit CB beginnt:

11001011 (Indexregister + Relativadresse)
00 f r

Rotations- oder Verschiebe-Befehle der Form f r
dabei bedeutet

f =		r =	
000	RLC	000	B
001	RRC	001	C
010	RL	010	D
011	RR	011	E

	100	SLA	100	H	
	101	SRA	101	L	
	111	SRL	110	(HL)	
	111	A			

11001011 (Indexregister + Relativadresse)
f b r

Bit-Manipulations-Befehle der Form f b,r

dabei bedeutet	f =	01	BIT	b = 000 0	r = 000	B
		10	RES	001 1	001	C
		11	SET	010 2	010	D
				011 3	011	E
				100 4	100	H
				101 5	101	L
				110 6	110	(HL)
				111 7	111	A

Als letztes noch die Befehle, deren Objekt-Code mit ED beginnt:

11101101
101 f 000

Befehle für blockweises Bewegen

dabei bedeutet	f =	00	LDI
		01	LDD
		10	LDIR
		11	LDDR

11101101
101 f 001

Such-Befehle

dabei bedeutet	f =	00	CPI
		01	CPD
		10	CPIR
		11	CPDR

11101101
101 f 010

Eingabe-Befehle

dabei bedeutet	f =	00	INI
		01	IND
		10	INIR
		11	INDR

11101101
1f011

Ausgabe-Befehle
dabei bedeutet f= 00 OUTI
01 OUTD
10 OTIR
11 OTDR

11101101
0110 f 111

Rotations-Befehle
dabei bedeutet f= 0 RRD
1 RLD

11101101
01 r 00 f

Ein-/Ausgabe-Befehle der Form IN r,(C) oder OUT (C),r
dabei bedeutet r = 000 B f= 0 IN r,(C)
001 C 1 OUT (C),r
010 D
011 E
100 H
101 L
111 A

11101101
01 dd 0011
yy
xx

16-Bit-Lade-Befehle der Form LD (xxyy),dd
dabei bedeutet dd = 00 BC
01 DE
10 HL
11 SP

11101101
01 dd 1011
yy
xx

16-Bit-Lade-Befehle der Form LD dd,(xxyy)
dabei bedeutet dd = 00 BC
01 DE
10 HL
11 SP

11101101
01 dd f 010

16-Bit-Arithmetik-Befehle der Form f HL,dd
dabei bedeutet dd = 00 BC
01 DE
10 HL
11 SP

f = 0 SBC
1 ADC

11101101
01000100

8-Bit-Arithmetik-Befehl NEG

11101101
0101 r 111

8-Bit-Lade-Befehle der Form LD A,r
dabei bedeutet r = 0 I
1 R

11101101
0100 r 111

8-Bit-Lade-Befehle der Form LD r,A
dabei bedeutet r = 0 I
1 R

11101101
0100 f 101

Unterprogramm-Rücksprung-Befehle
dabei bedeutet f = 0 RETN
1 RETI

11101101
010 p 110

Unterbrechungs-Modus-Befehle der Form IM p
dabei bedeutet p = 00 0
10 1
11 2

Anhang E

Pseudo-Operationen

Der Standard-Z80-Assembler von ZILOG erkennt die folgenden Pseudo-Operationen:

DEFB	define byte	Byte initialisieren
DEFW	define word	Wort initialisieren
DEFM	define string	Zeichenkette initialisieren
DEFS	define storage	Speicherplatz reservieren
EQU	equate	Konstante vereinbaren
ORG	Origin	Anfangsadresse für Code und Daten
END	end	Programmende

Anhang F

Kompatibilität des Prozessors 8080 mit dem Prozessor Z80

Kompatibilität

Unter der »Kompatibilität« eines Systems mit einem anderen wird im allgemeinen eine gewisse Gleichheit der beiden Systeme verstanden. Zum Beispiel sollten Programme, die für das System X geschrieben wurden, auch auf einem zu X »kompatiblen« System laufen. Es gibt in diesem Zusammenhang auch den Begriff »aufwärtskompatibel«. Dies bedeutet, daß ein neues System zu einem alten kompatibel ist, aber nicht umgekehrt, das heißt Programme, die für das alte System geschrieben wurden, sind auf dem neuen lauffähig, Programme für das neue System dagegen laufen nicht unbedingt auf dem alten.

In diesem Sinne ist unser Z80 dann »aufwärtskompatibel« zum 8080 von Intel, da:

1. alle 8080 Maschineninstruktionen beim Z80 auch vorhanden sind (vergleiche Tabelle F.2)
2. alle 8080-Register auch ein Pendant beim Z80 besitzen (vergleiche Tabelle F.1)
3. praktisch 99% der 8080-Programme auch auf dem Z80 laufen
4. der Z80 einen gegenüber dem 8080 erheblich erweiterten Befehlssatz besitzt (vergleiche Tabelle F.3), und somit Z80-Programme im allgemeinen nicht auf dem 8080 laufen.

Dabei ist zu beachten, daß sich die obig aufgeführten Argumente nur auf die Maschineninstruktionen beziehen (Objekt-Code), nicht aber auf den Source-Code (Mnemonics)!

Unterschiede

Es gibt einige kleine Unterschiede, die bei der Konvertierung von Programmen zu beachten sind:

1. Der Z80 benützt das *P-Flag* um einen Überlauf nach arithmetischen Operationen zu kennzeichnen. Der 8080 benützt dieses nur, um die Parität kennzuzeichnen.
2. Die *DAA*-Instruktion wird von den beiden Prozessoren unterschiedlich ausgeführt. Während die Korrektur vom Z80 bei der Addition und Subtraktion ausgeführt wird, wird sie vom 8080 nur bei der Addition durchgeführt.
3. Bei den Rotationsbefehlen des Z80 wird das *H-Flag* gelöscht, bei denen des 8080 nicht.
4. Wie aus der Tabelle F.1 ersichtlich, gibt es beim 8080 kein dem *N-Flag* entsprechendes Flag. Das korrespondierende Bit im *F-Register* stellt stets die logische »1« dar.

Es dürfte auch klar sein, daß die Ausführungszeiten bei den Prozessoren nicht identisch sind. Aus diesem Grund sind Programme für zeitkritische Vorgänge grundsätzlich auf dem dafür eingesetzten Prozessor zu entwickeln.

Zu den über die 8080-Eigenschaften hinausgehenden Eigenheiten des Prozessors 8085 von Intel ist der Z80 nicht kompatibel.

Tabellen

Tabelle F.1. *Registervergleich Z80 - 8080*

Z80-Register	8080-Register
A	A
A'	–
B	B
B'	–
C	C
C'	–
D	D
D'	–
E	E
E'	–
F	F
F'	–
H	H
H'	–
I	–
IX	–
IY	–
L	L
L'	–
R	–
PC	PC
SP	SP

Z80-Registerpaare	8080-Registerpaare
BC	B
DE	D
HL	H
AF	PSW
Z80 – indirektes Laden	**8080 – indirektes Laden**
(HL)	M
Z80-Flags	**8080-Flags**
C (carry)	C (carry)
H (half-carry)	AC (auxiliary-carry)
N (subtract)	–
P/O(parity/overflow)	P (parity)
S (sign)	S (sign)
Z (zero)	Z (zero)

Tabelle F.2. *Befehlsvergleich 8080 - Z80 (mnemonics)*

8080		Z80	
ACI	xx	ADC	A,xx
ADC	reg od. M	ADC	A,reg od. (HL)
ADD	reg od. M	ADD	A,reg od. (HL)
ADI	xx	ADD	A,xx
ANA	reg od. M	AND	reg od. (HL)
ANI	xx	AND	xx
CALL	xxyy	CALL	xxyy
CC	xxyy	CALL	C,xxyy
CM	xxyy	CALL	M,xxyy
CMA		CPL	
CMC		CCF	
CMP	reg od. M	CP	reg od. (HL)
CNC	xxyy	CALL	NC,xxyy
CNZ	xyy	CALL	NZ,xxyy
CP	xxyy	CALL	P,xxyy
CPE	xxyy	CALL	PE,xxyy
CPI	xx	CP	xx
CPO	xxyy	CALL	PO,xxyy
CZ	xxyy	CALL	Z,xxyy

8080		Z80	
DAA		DAA	
DAD	rp od. SP	ADD	HL,rp od. SP
DCR	reg od. M	DEC	reg od. (HL)
DCX	rp od. SP	DEC	rp od. SP
DI		DI	
EI		EI	
HLT		HALT	
IN	xx	IN	A,(xx)
INR	reg od. M	INC	reg od. (HL)
INX	rp od. SP	INC	rp od. SP
JC	xxyy	JP	C,xxyy
JM	xxyy	JP	M,xxyy
JMP	xxyy	JP	xxyy
JNC	xxyy	JP	NC,xxyy
JNZ	xxyy	JP	NZ,xxyy
JP	xxyy	JP	P,xxyy
JPE	xxyy	JP	PE,xxyy
JPO	xxyy	JP	PO,xxyy
JZ	xxyy	JP	Z,xxyy
LDA	xxyy	LD	A,(xxyy)
LDAX	B od.	D	LD A,(BC) od. (DE)
LHLD	xxyy	LD	HL,(xxyy)
LXI	rp od. SP,xxyy	LD	rp od. SP,xxyy
MOV	reg,reg od. M	LD	reg,reg od. (HL)
MOV	reg od. M,reg	LD	reg od. (HL),reg
MVI	reg od. M,xx	LD	reg od. (HL),xx
NOP		NOP	
ORA	reg od. M	OR	reg od. (HL)
ORI	xx	OR	xx
OUT	xx	OUT	(xx),A
PCHL		JP	(HL)
POP	rp od. PSW	POP	rp od. AF
PUSH	rp od. PSW	PUSH	rp od. AF
RAL		RLA	
RAR		RRA	
RC		RET	C
RET		RET	
RLC		RLCA	
RM		RET	M
RNC		RET	NC

8080		Z80	
RNZ		RET	NZ
RP		RET	P
RPE		RET	PE
RPO		RET	PO
RRC		RRCA	
RST	n	RST	n
RZ		RET	Z
SBB	reg od. M	SBC	A,reg od. (HL)
SBI	xx	SBC	A,xx
SHLD	xxyy	LD	(xxyy),HL
SPHL		LD	SP,HL
STA	xxyy	LD	(xxyy),A
STAX	B od. D	LD	(BC) od. (DE),A
STC		SCF	
SUB	reg od. M	SUB	reg od. (HL)
SUI	xx	SUB	xx
XCHG		EX	DE,HL
XRA	reg od. M	XOR	reg od. (HL)
XRI	xx	XOR	xx
XTHL		EX	(SP),HL

Hierbei bedeuten: reg ein 8-Bit-Register
rp ein Registerpaar

Tabelle F.3. *Befehlsvergleich Z80 - 8080 (mnemonics)*

Z80		8080	
ADC	A,xx	ACI	xx
ADC	A,(HL)	ADC	M
ADC	A,reg	ADC	reg
ADD	A,xx	ADI	xx
ADD	A,(HL)	ADD	M
ADD	A,reg	ADD	reg
ADD	HL,rp od. SP	DAD	rp od. SP
AND	xx	ANI	xx
AND	(HL)	ANA	M
AND	reg	ANA	reg
CALL	xxyy	CALL	xxyy

Z80		8080	
CALL	C,xxyy	CC	xxyy
CALL	M,xxyy	CM	xxyy
CALL	NC,xxyy	CNC	xxyy
CALL	NZ,xxyy	CNZ	xxyy
CALL	P,xxyy	CP	xxyy
CALL	PE,xxyy	CPE	xxyy
CALL	PO,xxyy	CPO	xxyy
CALL	Z,xxyy	CZ	xxyy
CCF		CMC	
CP	xx	CPI	xx
CP	(HL)	CMP	M
CP	reg	CMP	reg
CPL		CMA	
DAA		DAA	
DEC	(HL)	DCR	M
DEC	reg	DCR	reg
DEC	rp od. SP	DCX	rp od. SP
DI		DI	
EI		EI	
EX	DE,HL	XCHG	
EX	(SP),HL	XTHL	
HALT		HLT	
IN	A,(xx)	IN	xx
INC	(HL)	INR	M
INC	reg	INR	reg
INC	rp od. SP	INX	rp
JP	xxyy	JMP	xxyy
JP	C,xxyy	JC	xxyy
JP	(HL)	PCHL	
JP	M,xxyy	JM	xxyy
JP	NC,xxyy	JNC	xxyy
JP	NZ,xxyy	JNZ	xxyy
JP	P,xxyy	JP	xxyy
JP	PE,xxyy	JPE	xxyy
JP	PO,xxyy	JPO	xxyy
JP	Z,xxyy	JZ	xxyy
LD	A,(xxyy)	LDA	xxyy
LD	A,(BC) od. (DE)	LDAX	B od. D
LD	(xxyy),A	STA	xxyy
LD	(xxyy),HL	SHLD	

Z80		8080	
LD	(BC) od. (DE),A	STAX	B od. D
LD	HL,(xxyy)	LHLD	xxyy
LD	(HL),xx	MVI	M,xx
LD	(HL),reg	MOV	M,reg
LD	reg,xx	MVI	reg,xx
LD	reg,(HL)	MOV	reg,M
LD	reg,reg	MOV	reg,reg
LD	rp od. SP,xxyy	LXI	rp od. SP,xxyy
LD	SP,HL	SPHL	
NOP		NOP	
OR	xx	ORI	xx
OR	(HL)	ORA	M
OR	reg	ORA	reg
OUT	(xx),A	OUT	xx
POP	rp od. AF	POP	rp od. PSW
PUSH	rp od. AF	PUSH	rp od. PSW
RET		RET	
RET	C	RC	
RET	M	RM	
RET	NC	RNC	
RET	NZ	RNZ	
RET	P	RP	
RET	PE	RPE	
RET	PO	RPO	
RET	Z	RZ	
RLA		RAL	
RLCA		RLC	
RRA		RAR	
RRCA		RRC	
RST	n	RST	n
SBC	A,xx	SBI	xx
SBC	A,(HL)	SBB	M
SBC	A,reg	SBB	reg
SCF	STC		
SUB	xx	SUI	xx
SUB	(HL)	SUB	M
SUB	reg	SUB	reg
XOR	xx	XRI	xx
XOR	(HL)	XRA	M
XOR	reg	XRA	reg

Hierbei gelten dieselben Abkürzungen wie in der vorherigen Tabelle. Bei der Aufstellung ist zu beachten, daß nur die Z80-Befehle aufgeführt wurden, die auch ein 8080-Äquivalent besitzen, das heißt wenn ein Befehl in der Tabelle nicht gefunden werden kann, gibt es dafür keinen 8080-Befehl.

Index

Protext

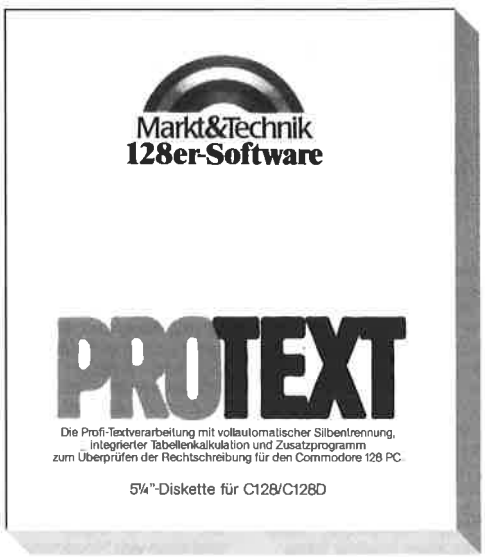
Die Profi-Textverarbeitung mit vollautomatischer Silbentrennung, integrierter Tabellenkalkulation und Zusatzprogramm zum Überprüfen der Rechtschreibung für den Commodore 128 PC.

Protext ist ein leicht bedienbares Textprogramm mit hoher Leistungsfähigkeit. Mit **Protext** sind daher auch Anfänger in der Lage, alle Vorteile eines professionellen Textprogramms zu nutzen. Überzeugen Sie sich selbst!

Was Protext alles kann:

- formatierte Ausgabe auf Bildschirm und Drucker mit programmierbaren Haltepunkten über serielle, V24- oder zwei Software-Centronics-Schnittstellen;
- vielfältige Formatanweisungen
- schnelle selbstlernende Textkorrektur mit deutschem (ca. 25000 Worte) Grundwortschatz sowie neun Kundenbibliotheken, die in Text umgewandelt, bearbeitet, ergänzt, sortiert und ausgedruckt werden können;
- Textübertragung per DFÜ,
- leistungsfähige Rechenmöglichkeiten mit Zeilenmarkierung (Rechentabulator), Kolonnenverarbeitung,

Hardware-Anforderungen: C128 oder C128D, 80-Zeichen-Monitor, Commodore-Drucker oder Drucker mit Centronics-Schnittstelle.



Bestell-Nr. 51254

DM 89,–*

(sFr 81,90/öS 718,–*)

* inkl. MwSt. Unverbindliche Preisempfehlung

Und dazu die weiterführende Literatur:

R. Schineis/H. Thies
Textverarbeitung mit Protext für den C128 PC.
1986, 258 Seiten
Eine systematische Einführung in die Bedienung von Protext. Druckersteuerung, Serienbrief, Tabellenkalkulation. Inklusive Druckertreiberdiskette.
Bestell-Nr. 90375
ISBN 3-89090-375-4
DM 39,–*
(sFr 35,90/öS 304,20*)

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

704325

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bücher zum Commodore 64/128

S. Vilsmeier
3D-Konstruktion mit GIGA-CAD Plus auf dem C64/C128
1986, 370 Seiten, inkl. 2 Disk.
Mit GIGA-CAD können Computergrafiken von besonderer Räumlichkeit und Faszination geschaffen werden. GIGA-CAD Plus ist schneller und einfacher zu bedienen, die Benutzeroberfläche wurde verbessert und der Befehlssatz erweitert. Die Eingabe erfolgt in erster Linie über den Joystick. Hardware-Anforderung: C64 mit Floppy 1541 oder C128 (im 64'er-Modus), Fernseher oder Monitor, Joystick und Commodore- oder Epson-kompatibler Drucker.
● Das verbesserte GIGA-CAD-Programm mit neuen Features wie erweitertem Befehlssatz und bis zu 10mal schneller liegt dem Buch im Floppy-1541-Format bei.
Best.-Nr. 90409
ISBN 3-89090-409-2
DM 49,–
(sFr 45,10/öS 382,20)

H. Haberl
Mini-CAD mit Hi-Eddi plus auf dem C64/C128
1986, 230 Seiten, inkl. Diskette
Auf der beiliegenden Diskette findet der Leser das vollständige Zeichenprogramm »Hi-Eddi«, mit dem das komfortable Erstellen von technischen Zeichnungen, Plänen oder Diagrammen ebenso möglich ist wie das Malen von farbigen Bildern, Entwurf und Ausdruck von Glückwunschkarten, Schildern, ja sogar von bewegten Sequenzen (kleine Trickfilme, Schaufenster-Werbung).
● Wer sagt, daß CAD auf dem C64 nicht möglich ist?!
Best.-Nr. 90136
ISBN 3-89090-136-0
DM 48,–
(sFr 44,20/öS 374,40)

B. Bornemann-Jeske
Vizawrite-Buch für den C64/C128
1987, 228 Seiten
Mit dem »Vizawrite-Buch« liegt erstmals ein vollständiges und detailliertes Arbeitsbuch für den Anfänger und den professionellen Anwender zur Textverarbeitung auf dem C64/C128 vor. Die Grundlagenkapitel führen Sie anhand kurzer Übungsaufgaben in die elementaren Funktionen des Systems ein. Das Kapitel für Fortgeschrittene zeigt Ihnen jede Programmfunktion im Detail. Zahlreiche praktische Tips aus verschiedenen Anwendungsbereichen ermöglichen Ihnen die optimale Nutzung Ihres Textverarbeitungssystems.
Best.-Nr. 90231
ISBN 3-89090-231-6
DM 49,–
(sFr 45,10/öS 382,20)

O. Hartwig
Experimente zur Künstlichen Intelligenz mit C64/C128
1987, 248 Seiten
Sind Maschinen intelligent? Können Computer denken? Erschließen Sie sich eines der interessantesten Gebiete der modernen Computerforschung! Anhand zahlreicher Programme erfahren Sie hier die Möglichkeiten der Künstlichen Intelligenz, speziell auf dem C64 und dem C128. Der Schwerpunkt des Buches liegt auf der Praxis. Alle KI-Techniken werden durch anschauliche Programme vorgestellt, die sofort nachvollziehbar sind. Zusätzlich erhalten Sie jede Menge Anregungen zu eigenen Experimenten. Die KI-Programme können ohne weiteres in eigene Programme integriert werden.
Best.-Nr. 90472
ISBN 3-89090-472-6
DM 49,–
(sFr 45,10/öS 382,20)

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

708358

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Version 2.41

dBASE II
für Commodore 128/128 D

dBASE II, das meistverkaufte Programm unter den Datenbanksystemen, gibt es jetzt im CP/M-Modus für den C 128. Es eröffnet Ihnen optimale Möglichkeiten der Daten- und Dateihandhabung. Einfach und schnell können Datenstrukturen definiert, benutzt und geändert werden. Der Datenzugriff erfolgt sequentiell oder nach frei wählbaren Kriterien, die integrierte Kommandosprache ermöglicht den Aufbau kompletter Anwendungen wie Finanzbuchhaltung, Lagerverwaltung, Betriebsabrechnung usw.

Lieferumfang:
- Originalhandbuch von Ashton-Tate
- Beschreibung der Commodore-128-PC-spezifischen Version

Hardware-Anforderungen:
Commodore 128 PC, Diskettenlaufwerk, 80-Zeichen-Monitor, beliebiger Commodore-Drucker oder ein Drucker mit Centronics-Schnittstelle über Userport

Markt&Technik
128er-Software

dBASE™ II
ASHTON-TATE
für den
Commodore 128 PC

5¼"-Diskette
im Floppy 1541-Format

Bestell-Nr. 50303

DM 199,–

(sFr 178,–/öS 1890,–*)
inkl. MwSt. Unverbindliche Preisempfehlung

Und dazu die weiterführende Literatur:

Dr. P. Albrecht
dBASE II für den Commodore 128 PC
Dieses klassische Einführungs- und Nachschlagewerk begleitet Sie mit nützlichen Hinweisen bei Ihrer täglichen Arbeit mit dBASE II.
Bestell-Nr. 90189,
ISBN 3-89090-189-1
DM 49,–
(sFr 45,10/öS 382,20)

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

704322

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613 0

Multiplan

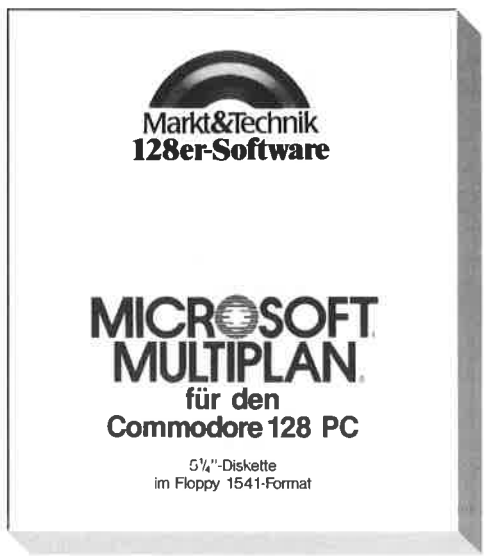
Version 1.06

für Commodore 128/128 D

Wenn Sie die zeitraubende manuelle Verwaltung tabellarischer Aufstellungen mit Bleistift, Radiergummi und Rechenmaschine satt haben, dann ist MULTIPLAN, das System zur Bearbeitung »elektronischer Datenblätter«, genau das richtige für Sie! Das benutzerfreundliche und leistungsfähige Tabellenkalkulationsprogramm kann bei allen Analyse- und Planungsberechnungen eingesetzt werden wie zum Beispiel Budgetplanungen, Produktkalkulationen, Personalkosten usw. Spezielle Formatierungs-, Aufbereitungs- und Druckanweisungen ermöglichen außerdem optimal aufbereitete Präsentationsunterlagen!

5¼"-Diskette für den Commodore 128 PC.

Hardware-Anforderungen: Commodore 128 PC, Diskettenlaufwerk, 80-Zeichen-Monitor, beliebiger Commodore-Drucker oder ein Drucker mit Centronics-Schnittstelle



Bestell-Nr. 50203

DM 199,–*

(sFr 178/öS 1890,–*)

* inkl. MwSt. Unverbindliche Preisempfehlung

Und dazu die weiterführende Literatur:

Dr. P. Albrecht
Multiplan für den Commodore 128 PC
1985, 226 Seiten
Mit diesem Buch werden Sie Ihre Tabellenkalkulation ohne Probleme in den Griff bekommen. Als Nachschlagewerk leistet es auch dem Profi nützliche Dienste.
Bestell-Nr. MT 836
ISBN 3-89090-187-5
DM 49,–
(sFr 45,10/öS 382,20)

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

705323

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Das Z80-BUCH

Dr. EBERHARD ZEHENDNER, geboren 1956, studierte bis zu seiner Diplomierung im Jahr 1982 Mathematik und Informatik an der Technischen Universität München. Seitdem ist er am Institut für Mathematik der Universität Augsburg tätig, derzeit als wissenschaftlicher Mitarbeiter am Lehrstuhl Informatik I. Seine Forschungsschwerpunkte sind die Codierungstheorie und die Rechnerarchitektur. Der frischgebackene Doktor der Naturwissenschaften konnte während seines Studiums und durch seine Lehrtätigkeit an der Universität intensiv Erfahrungen mit verschiedenen Programmiersprachen sammeln und konzentriert seine Aktivitäten zur Zeit auf die Ausbildung von Studenten im Umgang mit Mikrocomputern.

Der Prozessor Z80 der Firma Zilog hat als CPU für CP/M-Computer eine enorme Verbreitung gefunden. Obwohl er eine relativ alte Entwicklung ist, wird er auch in neueste Systeme wie zum Beispiel die erfolgreichen Computer der Firma Schneider eingebaut; in Interface-Karten oder Peripheriegeräten wie Plottern und Druckern findet man oft einen Z80.

Als billiger und in seiner Leistung ständig verbesserter Prozessor eignet sich der Z80 besonders für den Selbstbau eines Computersystems.

Die Programmierung des Z80 ist an sich nicht schwer, sollte jedoch von Anfang an systematisch gelernt werden. Das vorliegende Buch wird für die Programmierausbildung an der Universität benutzt; es bietet eine problemorientierte Vorgehensweise, die für Anfänger wie Fortgeschrittene gleichermaßen geeignet ist. Für wichtige, in der Praxis immer wieder auftretende Probleme werden sichere Standardlösungen angeboten. Die Orientierung an konkreten Anwendungen und der systematische, schrittweise Aufbau ermöglichen auch dem interessierten Laien einen Einstieg in die Assemblerprogrammierung im Selbststudium.

Wer dieses Buch gelesen hat, wird in der Lage sein, schnell, zuverlässig und verständlich zu programmieren. Eine wertvolle Hilfe für den Programmierer ist der umfangreiche Anhang. Hier findet er vollständige Listen der Z80-Assemblerbefehle jeweils sortiert nach Funktionsgruppen, Bedeutung und Objekt-Codes, eine Darstellung des systematischen Aufbaus des Befehlssatzes, einen Überblick über die Pseudo-Operationen und eine Gegenüberstellung der Z80-Befehle mit denen des 8080.

ISB N 3-89090-219-7

Markt&Technik

4 001057 902190

DM 59,–
sFr. 54,30
öS 460,20