Die PC Assemblersprache

Paul A. Carter

15. Dezember 2006

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Vorwort

Ziel

Das Ziel dieses Buches besteht darin, dem Leser ein besseres Verständnis darüber zu geben, wie Computer auf einem niedrigeren Level als in Programmiersprachen wie Pascal wirklich arbeiten. Durch den Erwerb eines tieferen Verständnisses wie Computer arbeiten, kann der Lesern oft sehr viel produktiver Software in Hochsprachen wie C und C++ entwickeln. Ein ausgezeichneter Weg, um dieses Ziel zu erreichen, ist, in Assembler programmieren zu lernen. Andere PC-Assemblerbücher lehren immer noch den 8086 Prozessor zu programmieren, den der originale PC 1981 benutzte! Die 8086 Prozessoren unterstützten nur den *real* Modus. In diesem Modus kann jedes Programm alle Speicherstellen oder Geräte im Computer ansprechen. Dieser Modus ist für ein sicheres Betriebssystem mit Multitasking nicht geeignet. Dagegen behandelt dieses Buch wie der 80386 und spätere Prozessoren im *protected* Modus programmiert werden (dem Modus, in dem Windows und Linux laufen). Dieser Modus unterstützt die Merkmale, die moderne Betriebssysteme erwarten, wie virtuellen Speicher und geschützten Speicher. Es gibt verschiedene Gründe den protected Mode zu verwenden:

1. Es ist einfacher im protected Mode zu programmieren als im 8086 real Mode, den andere Bücher verwenden.
2. Alle modernen PC-Betriebssysteme laufen im protected Mode.
3. Es gibt freie Software, die in diesem Modus läuft.

Das Fehlen von Lehrbüchern für die Assemblerprogrammierung des PC im protected Mode ist der Hauptgrund, dass der Autor dieses Buch schrieb.

Wie oben angedeutet, macht dieses Buch Gebrauch von Free/Open Source Software: nämlich dem NASM Assembler und dem DJGPP C/C++ Compiler. Beide stehen zum Download im Internet zur Verfügung. Der Text bespricht außerdem, wie der Assemblercode von NASM unter Linux und mit den C/C++ Compilern von Borland und Microsoft unter Windows verwendet werden kann. Beispiele für alle diese Plattformen können auf meiner Webseite gefunden werden: `http://www.drpaulcarter.com/pcasm`. Sie *müssen* den Beispielcode herunterladen, wenn Sie viele der Beispiele in diesem Tutorial assemblieren und laufen lassen möchten.

Sind Sie sich darüber bewusst, dass dieser Text nicht versucht, jeden Aspekt der Assemblerprogrammierung abzudecken. Der Autor hat versucht, die wichtigsten Themen abzudecken, mit denen *alle* Programmierer bekannt sein sollten.

Danksagungen

Der Autor möchte den vielen Programmierern auf der Welt danken, die zur Free/Open Source Bewegung beigetragen haben. All die Programme und sogar dieses Buch selbst wurden unter Verwendung freier Software produziert. Besonders möchte der Autor John S. Fine, Simon Tatham, Julian Hall und anderen für die Entwicklung des NASM Assemblers danken, auf dem alle Beispiele in diesem Buch basieren; DJ Delorie für die Entwicklung des verwendeten DJGPP C/C++ Compilers; den zahlreichen Personen, die zum GNU gcc Compiler beigetragen haben, auf dem DJGPP beruht; Donald E. Knuth und anderen für die Entwicklung der TEX and LATEX 2_ε Satzsprachen, die benutzt wurden, um diesen Buch zu produzieren; Richard Stallman (Gründer der Free Software Foundation), Linus Torvalds (Schöpfer des Linux Kernels) und anderen, die die zugrunde liegende Software produziert haben, die der Autor benutzte, um dieses Werk zu produzieren.

Dank gebührt den folgenden Personen für Korrekturen:

- John S. Fine
- Marcelo Henrique Pinto de Almeida
- Sam Hopkins
- Nick D'Imperio
- Jeremiah Lawrence
- Ed Beroset
- Jerry Gembarowski
- Ziqiang Peng
- Eno Compton
- Josh I Cates
- Mik Mifflin
- Luke Wallis
- Gaku Ueda
- Brian Heward
- Chad Gorshing
- F. Gotti
- Bob Wilkinson
- Markus Koegel
- Louis Taber
- Dave Kiddell
- Eduardo Horowitz
- Sébastien Le Ray
- Nehal Mistry

Quellen im Internet[1]

Die Seite des Authors	`http://www.drpaulcarter.com/`
NASM SourceForge Seite	`http://sourceforge.net/projects/nasm/`
DJGPP	`http://www.delorie.com/djgpp`
Linux Assembly	`http://www.linuxassembly.org/`
The Art of Assembly	`http://webster.cs.ucr.edu/`
USENET	`comp.lang.asm.x86`
Intel Dokumentation	`http://developer.intel.com/design/Pentium4/documentation.htm`

Feedback

Der Autor begrüßt jedes Feedback über dieses Werk.

E-mail: `pacman128@gmail.com`
WWW: `http://www.drpaulcarter.com/pcasm`

[1] Stand 2006-12-15 [Anm. d. Ü.]

Kapitel 1

Einführung

1.1 Zahlensysteme

Speicher in einem Computer enthält Zahlen. Computer speichern diese Zahlen nicht dezimal (Basis 10). Weil es die Hardware stark vereinfacht, speichern Computer alle Informationen in einem binären (Basis 2) Format. Wiederholen wir zunächst das Dezimalsystem.

1.1.1 Dezimal

Zahlen zur Basis 10 sind aus 10 möglichen Ziffern (0-9) zusammengesetzt. Jeder Ziffer einer Zahl ist eine Potenz von 10 zugeordnet, die auf ihrer Position in der Zahl beruht. Zum Beispiel:

$$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

1.1.2 Binär

Zahlen zur Basis 2 sind aus 2 möglichen Ziffern (0 und 1) zusammengesetzt. Jeder Ziffer einer Zahl ist eine Potenz von 2 zugeordnet, die auf ihrer Position in der Zahl beruht. (Eine einzelne binäre Ziffer wird *Bit* genannt.) Zum Beispiel:

$$\begin{aligned} 11001_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 1 \\ &= 25 \end{aligned}$$

Dies zeigt, wie binär in dezimal umgewandelt werden kann. Tabelle 1.1 zeigt, wie die ersten paar Zahlen im Binären dargestellt werden.

Abbildung 1.1 zeigt, wie einzelne binäre Ziffern (d. h. Bits) addiert werden.

Wenn man die folgende dezimale Division betrachtet:

$$1234 \div 10 = 123\ r\ 4$$

kann man sehen, dass diese Division die am weitesten rechts stehende dezimale Ziffer der Zahl entfernt und die anderen dezimalen Ziffern eine Stelle nach rechts

Dezimal	Binär		Dezimal	Binär
0	0000		8	1000
1	0001		9	1001
2	0010		10	1010
3	0011		11	1011
4	0100		12	1100
5	0101		13	1101
6	0110		14	1110
7	0111		15	1111

Tabelle 1.1: 0 bis 15 in dezimal und binär

Ohne vorherigem Übertrag				Mit vorherigem Übertrag			
0	0	1	1	0	0	1	1
+0	+1	+0	+1	+0	+1	+0	+1
0	1	1	0	1	0	0	1
			c		c	c	c

Abbildung 1.1: Binäre Addition (c steht für *Übertrag*)

rückt. Division durch zwei führt eine ähnliche Operation durch, nur für die binären Ziffern der Zahl. Betrachten wir dazu die folgende binäre Division:[1]

$$1101_2 \div 10_2 = 110_2 \; r \; 1$$

Diese Tatsache kann benutzt werden, um eine dezimale Zahl in ihre binäre Darstellung umzuwandeln, wie Abbildung 1.2 zeigt. Diese Methode findet die am weitesten rechts stehende Ziffer zuerst, diese Ziffer wird das *niederwertigste Bit* (*least significant bit*, lsb) genannt. Die am weitesten links stehende Ziffer wird das *höchstwertige Bit* (*most significant bit*, msb) genannt. Die grundlegende Speichereinheit besteht aus 8 Bits und wird *Byte* genannt.

Dezimal	Binär
$25 \div 2 = 12 \; r \; 1$	$11001 \div 10 = 1100 \; r \; 1$
$12 \div 2 = 6 \; r \; 0$	$1100 \div 10 = 110 \; r \; 0$
$6 \div 2 = 3 \; r \; 0$	$110 \div 10 = 11 \; r \; 0$
$3 \div 2 = 1 \; r \; 1$	$11 \div 10 = 1 \; r \; 1$
$1 \div 2 = 0 \; r \; 1$	$1 \div 10 = 0 \; r \; 1$

folglich $25_{10} = 11001_2$

Abbildung 1.2: Umwandlung von dezimal nach binär

[1] Die tief gestellte 2 wird benutzt, um zu zeigen, dass die Zahl in binär und nicht in dezimal dargestellt wird.

1.1.3 Hexadezimal

Hexadezimale Zahlen benutzen die Basis 16. Hexadezimal (oder kurz *hex*) kann als Abkürzung für binäre Zahlen verwendet werden. Hex hat 16 mögliche Ziffern. Dies wirft ein Problem auf, da es für die zusätzlichen Ziffern nach der 9 keine Symbole gibt. Per Konvention werden Buchstaben für diese zusätzlichen Ziffern verwendet. Die 16 Hexziffern sind 0-9, dann A, B, C, D, E und F. Die Ziffer A ist äquivalent zu 10 in dezimal, B ist 11, usw. Jeder Ziffer einer Hexzahl ist eine Potenz von 16 zugeordnet. Beispiel:

$$\begin{aligned} 2\mathrm{BD}_{16} &= 2 \times 16^2 + 11 \times 16^1 + 13 \times 16^0 \\ &= 512 + 176 + 13 \\ &= 701 \end{aligned}$$

Um von dezimal nach hex zu wandeln, benutzen wir die gleiche Idee, die für die binäre Konversion verwendet wurde, außer eben durch 16 zu teilen. Für ein Beispiel siehe Abbildung 1.3.

$$\begin{aligned} 589 \div 16 &= 36\ r\ 13 \\ 36 \div 16 &= 2\ r\ 4 \\ 2 \div 16 &= 0\ r\ 2 \end{aligned}$$

folglich $589 = 24\mathrm{D}_{16}$

Abbildung 1.3: Umwandlung von dezimal nach hexadezimal

Der Grund, dass hex nützlich ist, ist, dass es einen sehr einfachen Weg gibt, um zwischen hex und binär zu wandeln. Binäre Zahlen werden schnell groß und unhandlich. Hex liefert einen wesentlich kompakteren Weg um binär darzustellen.

Um eine Hexzahl nach binär zu wandeln, konvertieren wir einfach jede Hexziffer zu einer 4 bit Binärzahl. Zum Beispiel wird $24\mathrm{D}_{16}$ zu $0010\ 0100\ 1101_2$ gewandelt. Beachten Sie, dass die führenden Nullen der 4 bit Zahlen wichtig sind! Wenn die führende Null für die mittlere Ziffer von $24\mathrm{D}_{16}$ weggelassen wird, ist das Ergebnis falsch. Die Wandlung von binär nach hex ist genauso einfach. Man führt den Prozess in umgekehrter Reihenfolge aus. Wandeln Sie alle 4 bit Segmente der Binärzahl nach hex. Beginnen Sie am rechten, nicht am linken Ende der Binärzahl. Dies stellt sicher, dass der Prozess die korrekten 4 bit Segmente benutzt.[2] Beispiel:

110	0000	0101	1010	0111	1110_2
6	0	5	A	7	E_{16}

[2]Wem es nicht klar ist, warum der Startpunkt einen Unterschied macht, versuche das Beispiel von links her zu wandeln.

Eine 4 bit Zahl wird ein *Nibble* genannt. Deshalb entspricht jede Hexziffer einem Nibble. Zwei Nibbles geben ein Byte und daher kann ein Byte durch eine 2-stellige Hexzahl dargestellt werden. Ein Bytewert reicht von 0 bis 11111111 in binär, 0 bis FF in hex und 0 bis 255 in dezimal.

1.2 Aufbau eines Computers

1.2.1 Speicher

Speicher wird gemessen in Einheiten von Kilobyte (2^{10} = 1 024 Byte), Megabyte (2^{20} = 1 048 576 Byte) und Gigabyte (2^{30} = 1 073 741 824 Byte).

Die grundlegende Speichereinheit ist ein Byte. Ein Computer mit 32 Megabyte Speicher kann ungefähr 32 Millionen Bytes an Informationen speichern. Jedes Byte im Speicher ist mit einer eindeutigen Zahl verbunden, die als Adresse bekannt ist, wie Abbildung 1.4 zeigt.

Adresse	0	1	2	3	4	5	6	7
Speicher	2A	45	B8	20	8F	CD	12	2E

Abbildung 1.4: Speicheradressen

Oft wird Speicher in größeren Einheiten als einzelne Bytes verwendet. In der PC-Architektur sind Namen für diese größeren Speichereinheiten vergeben worden, die Tabelle 1.2 zeigt.

word	2 Byte
double word	4 Byte
quad word	8 Byte
paragraph	16 Byte

Tabelle 1.2: Speichereinheiten

Alle Daten im Speicher sind numerisch. Zeichen werden gespeichert, indem ein *Zeichencode* verwendet wird, der Zahlen auf Zeichen abbildet. Einer der häufigsten Zeichencodes ist als *ASCII* (American Standard Code for Information Interchange) bekannt. Ein neuer, vollständigerer Code, der ASCII verdrängt, ist Unicode. Ein hauptsächlicher Unterschied zwischen den beiden Codes ist, dass ASCII ein Byte benutzt, um ein Zeichen zu kodieren, während Unicode zwei Byte (oder ein *Wort*) pro Zeichen benutzt. Zum Beispiel bildet ASCII das Byte 41_{16} (65_{10}) auf das Zeichen *A* ab; Unicode das Wort 0041_{16}. Da ASCII ein Byte benutzt, ist es auf nur 256 verschiedene Zeichen beschränkt.[3] Unicode weitet die ASCII-Werte auf Wörter aus und erlaubt so die Repräsentation von wesentlich mehr Zeichen. Dies ist wichtig, um Zeichen für alle Sprachen der Welt zu repräsentieren.

1.2.2 Die CPU

Die Zentraleinheit (Central Processing Unit, CPU) ist das physikalische Gerät, das Befehle ausführt. Die Befehle, die CPUs ausführen, sind in der Regel sehr

[3] In Wirklichkeit verwendet ASCII nur die unteren 7 Bits und kann so nur 128 verschiedene Werte benutzen.

einfach. Befehle können erfordern, dass Daten, die sie verarbeiten, in speziellen Speichereinheiten innerhalb der CPU selbst, die *Register* genannt werden, liegen müssen. Die CPU kann auf Daten in Registern viel schneller zugreifen, als auf Daten im Speicher. Jedoch ist die Zahl der Register in einer CPU begrenzt, sodass der Programmierer dafür sorgen muss, nur gerade benötigte Daten in Registern zu halten.

Die Befehle, die ein CPU-Typ ausführen kann, bilden die *Maschinensprache* dieser CPU. Maschinenprogramme haben eine viel einfachere Struktur als Hochsprachen. Befehle in Maschinensprache werden als nackte Zahlen kodiert, nicht in freundlichen Textformaten. Um effizient zu laufen, muss eine CPU fähig sein, den Zweck einer Instruktion sehr schnell zu dekodieren. Maschinensprache wird mit diesem Ziel entwickelt und nicht, um leicht durch Menschen entziffert werden zu können. Programme in anderen Sprachen müssen in die native Maschinensprache der CPU konvertiert werden, um auf einem Computer zu laufen. Ein *Compiler* ist ein Programm, das Programme, die in einer Programmiersprache geschrieben sind, in die Maschinensprache einer bestimmten Computerarchitektur übersetzt. Ganz allgemein hat jeder CPU-Typ seine eigene einzigartige Maschinensprache. Das ist ein Grund, warum Programme, die für einen Mac geschrieben wurden, auf einem IBM-typischen PC nicht laufen können.

Computer benutzen einen *Taktgeber*, um die Ausführung der Instruktionen zu synchronisieren. Der Taktgeber liefert Impulse mit einer festgelegten Frequenz (als *Taktfrequenz* bezeichnet). Wenn man einen 1.5 GHz Computer kauft, ist 1.5 GHz die Frequenz dieses Taktgebers. Die Elektronik der CPU benutzt den Takt um ihre Operationen korrekt durchzuführen, so, wie die Schläge eines Metronoms einem helfen, Musik mit dem korrekten Rhythmus zu spielen. Die Anzahl der Schläge (oder *Taktzyklen*, wie sie gewöhnlich genannt werden), die eine Instruktion benötigt, hängt von CPU-Generation und -Modell ab. Die Anzahl der Zyklen hängt von den Instruktionen vor ihr und ebenso von anderen Faktoren ab.

GHz *steht für Gigahertz oder eine Milliarde Zyklen pro Sekunde. Eine 1.5 GHz CPU erhält 1.5 Milliarden Taktimpulse pro Sekunde.*

1.2.3 Die 80x86 CPU-Familie

IBM-Typ PCs enthalten eine CPU aus Intels 80x86 Familie (oder einen Klon davon). Die CPUs in dieser Familie haben alle einige gemeinsame Merkmale, die eine grundlegende Maschinensprache einschließen. In den neueren Mitgliedern wurden diese Merkmale jedoch stark erweitert.

8086, 8088: Diese CPUs sind vom Standpunkt der Programmierung identisch. Es sind die CPUs, die in den frühesten PCs verwendet wurden. Sie stellen verschiedene 16 bit Register zur Verfügung: AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, SS, ES, IP, FLAGS. Sie unterstützen Speicher nur bis zu einem Megabyte und arbeiten nur im *real Mode*. In diesem Modus kann ein Programm jede Speicheradresse ansprechen, selbst den Speicher anderer Programme! Das macht Fehlersuche und Sicherheit sehr schwierig! Ebenso müssen Programme in *Segmente* unterteilt werden. Jedes Segment kann nicht größer als 64K werden.

80286: Diese CPU wurde in PCs der AT-Klasse verwendet. Sie fügt zur grundlegenden Maschinensprache der 8086/8088 einige neue Instruktionen hinzu. Jedoch ist ihr Hauptmerkmal der *16-bit protected Mode*. In diesem Modus kann sie auf bis zu 16 Megabyte zugreifen und schützt Programme

davor, gegenseitig auf ihre Speicherbereiche zuzugreifen. Jedoch werden Programme immer noch in Segmente unterteilt, die nicht größer als 64K sein können.

80386: Diese CPU hat die 80286 stark erweitert. Zuerst dehnt sie viele der Register auf 32 bit aus (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP, EFLAGS) und fügt zwei neue 16 bit Register, FS und GS, hinzu. Sie führt auch einen neuen *32-bit protected Mode* ein. In diesem Modus kann sie auf bis zu 4 Gigabyte zugreifen. Programme werden wieder in Segmente eingeteilt, aber nun kann jedes Segment ebenso bis zu 4 Gigabyte groß sein!

80486/Pentium/Pentium Pro: Diese Mitglieder der 80x86 Familie fügen nur wenig neue Merkmale hinzu. Sie beschleunigen hauptsächlich die Ausführung der Befehle.

Pentium MMX: Dieser Prozessor fügt dem Pentium die MMX (MultiMedia eXtension) Befehle hinzu. Diese Befehle können allgemeine grafische Operationen beschleunigen.

Pentium II: Das ist der Pentium Pro Prozessor mit dem MMX Befehlssatz. (Der Pentium III ist im Wesentlichen nur ein schnellerer Pentium II.)

1.2.4 16 bit Register der 8086

Die ursprüngliche 8086 CPU besaß vier 16 bit Allzweckregister: AX, BX, CX, DX. Jedes dieser Register konnte in zwei 8 bit Register aufgeteilt werden. Zum Beispiel konnte das AX-Register in die Register AH und AL aufgeteilt werden, wie Abbildung 1.5 zeigt. Das AH Register enthält die oberen (oder hohen) 8 Bits von AX, und AL enthält die unteren 8 Bits von AX. Oft werden AH und AL als unabhängige ein-Byte Register benutzt; jedoch ist es wichtig zu realisieren, dass sie nicht unabhängig von AX sind. Eine Änderung des Wertes von AX wird AH und AL ändern und umgekehrt. Die Allzweckregister werden in vielen Datenbewegungen und arithmetischen Befehlen verwendet.

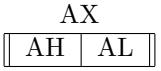


Abbildung 1.5: Das AX Register

Es gibt zwei 16 bit Indexregister: SI und DI. Sie werden oft als Zeiger verwendet, können aber für viele Zwecke genauso wie die allgemeinen Register verwendet werden. Jedoch können sie nicht in 8 bit Register aufgeteilt werden.

Die 16 bit Register BP und SP werden als Zeiger auf Daten im Stack der Maschinensprache verwendet und werden Base Pointer bzw. Stack Pointer genannt. Sie werden später noch besprochen.

Die 16 bit Register CS, DS, SS, und ES sind *Segmentregister*. Sie bestimmen, welcher Speicher für die verschiedenen Teile eines Programms benutzt wird. CS steht für Code Segment, DS für Daten Segment, SS für Stack Segment und ES für Extra Segment. ES wird als temporäres Segmentregister verwendet. Die Details zu diesen Registern finden sich in den Abschnitten 1.2.6 und 1.2.7.

Das Instruction Pointer (IP) Register wird zusammen mit dem CS Register benutzt, um sich die Adresse des nächsten durch die CPU auszuführenden Befehls zu merken. Sobald ein Befehl zur Ausführung kommt, wird normalerweise IP hochgezählt, um auf den nächsten Befehl im Speicher zu zeigen.

Das FLAGS Register speichert wichtige Informationen über das Ergebnis eines vorherigen Befehls. Diese Ergebnisse werden als einzelne Bits im Register gespeichert. Zum Beispiel ist das Z-Bit 1, wenn das Ergebnis des vorherigen Befehls Null war oder 0, wenn es nicht Null war. Nicht alle Befehle verändern die Bits in FLAGS; ziehen Sie die Tabelle im Anhang zu Rate, um zu sehen, wie die einzelnen Befehle das FLAGS Register beeinflussen.

1.2.5 32 bit Register der 80386

Der 80386 und spätere Prozessoren besitzen erweiterte Register. Zum Beispiel wurde das 16 bit AX Register auf 32 bit erweitert. Um abwärts kompatibel zu sein, bezieht sich AX immer noch auf das 16 bit Register und EAX wird verwendet, um sich auf das erweiterte 32 bit Register zu beziehen. AX sind die unteren 16 Bits von EAX genauso wie AL die unteren 8 Bits von AX (und EAX) sind. Es gibt keine Möglichkeit, direkt auf die oberen 16 Bits von EAX zuzugreifen. Die anderen erweiterten Register sind EBX, ECX, EDX, ESI und EDI.

Viele der anderen Register wurden ebenfalls erweitert. BP wird zu EBP; SP zu ESP; FLAGS zu EFLAGS und IP zu EIP. Jedoch werden im Gegensatz zu den Index- und allgemeinen Registern im 32-bit protected Mode (weiter unten besprochen) nur die erweiterten Versionen dieser Register benutzt.

Die Segmentregister sind in der 80386 immer noch 16 bit. Es gibt auch zwei neue Segmentregister: FS und GS. Ihre Namen stehen für nichts Bestimmtes. Sie sind zusätzliche temporäre Segmentregister (wie ES).

Eine Definition des Begriffs *word* bezieht sich auf die Größe der Datenregister der CPU. In der 80x86 Familie ist der Begriff nun etwas verwirrend. Aus Tabelle 1.2 kann man entnehmen, dass *word* als 2 Byte (16 bit) definiert ist. Diese Bedeutung wurde vergeben, als die 8086 zuerst herauskam. Als die 80386 entwickelt wurde, wurde beschlossen, die Definition von *word* nicht zu ändern, obwohl die Größe des Registers geändert wurde.

1.2.6 Real Mode

Im real Mode ist der Speicher auf nur ein Megabyte (2^{20} Byte) beschränkt. Gültige Adressen reichen (in hex) von 00000 bis FFFFF. Diese Adressen benötigen eine 20 bit Zahl. Offensichtlich passt eine 20 bit Zahl in keines der 16 bit Register der 8086. Intel löste das Problem, indem sie zwei 16 bit Werte benutzen, um eine Adresse festzulegen. Der erste 16 bit Wert wird *Segment* genannt. Segmentwerte müssen in Segmentregistern gespeichert werden. Der zweite 16 bit Wert wird *Offset* genannt. Die physikalische Adresse, die durch ein 32 bit *Segment:Offset* Paar bezeichnet wird, wird berechnet nach der Formel

So, woher stammt die berüchtigte DOS 640K Grenze? Das BIOS belegte etwas von dem 1M für seinen Code und für Hardwaregeräte wie dem Videospeicher.

$$16 \times \text{Segment} + \text{Offset}$$

Mit 16 in hex zu multiplizieren ist einfach, nur eine 0 rechts an die Zahl anhängen. Zum Beispiel ist die physikalische Adresse, die durch 047C:0048 angesprochen wird, gegeben durch:

$$\begin{array}{r} 047C0 \\ +0048 \\ \hline 04808 \end{array}$$

Im Effekt ist der Segmentwert eine Paragraphennummer (siehe Tabelle 1.2).

Segmentierte Adressen im real Mode haben Nachteile:

- Ein einzelner Segmentwert kann nur 64K Speicher referenzieren (die Obergrenze des 16 bit Offsets). Was ist mit einem Programm, das mehr als 64K Code besitzt? Ein einziger Wert in CS kann nicht für die gesamte Ausführung des Programms benutzt werden. Das Programm muss in Abschnitte (*Segmente* genannt) aufgeteilt werden, die weniger als 64K groß sind. Wenn die Ausführung von einem zu einem anderen Segment wechselt, muss der Wert von CS geändert werden. Ähnliche Probleme treten mit großen Datenmengen und dem DS Register auf. Das kann sehr lästig sein!
- Kein Byte im Speicher hat eine eindeutige Adresse. Auf die physikalische Adresse 04808 kann mit 047C:0048, 047D:0038, 047E:0028 oder 047B:0058 zugegriffen werden. Dies kann den Vergleich von segmentierten Adressen kompliziert machen.

1.2.7 16-bit Protected Mode

Im protected Mode der 80286 werden Selektor-Werte völlig anders als im real Mode interpretiert.[4] Im real Mode ist ein Segment-Wert eine Paragraphennummer des physikalischen Speichers. Im protected Mode ist ein Selektorwert ein *Index* in eine *Deskriptorentabelle*. In beiden Modi werden Programme in Segmente geteilt. Im real Mode befinden sich diese Segmente an festen Stellen im physikalischen Speicher und der Segmentwert bezeichnet die Paragraphennummer des Anfangs des Segments. Im protected Mode sind die Segmente nicht an festgelegten Positionen im physikalischen Speicher. Tatsächlich müssen sie nicht einmal im Speicher sein!

Protected Mode ermöglicht eine Technik, die *virtueller Speicher* genannt wird. Die grundlegende Idee hinter einem virtuellen Speichersystem ist, nur Daten und Code im Speicher zu halten, die das Programm gerade benutzt. Andere Daten und Code werden temporär auf der Disk gespeichert, bis sie wieder benötigt werden. Im 16-bit protected Mode werden Segmente zwischen Speicher und Disk verschoben, wie sie gerade benötigt werden. Wenn ein Segment von der Disk zurück in den Speicher geladen wird, ist es sehr wahrscheinlich, dass es in einen anderen Speicherbereich geladen wird als es war, bevor es auf die Disk kopiert wurde. All dies wird transparent vom Betriebssystem durchgeführt. Das Programm muss nicht anders geschrieben werden damit virtueller Speicher funktioniert.

Im protected Mode ist jedem Segment ein Eintrag in einer Deskriptorentabelle zugeordnet. Dieser Eintrag enthält all die Informationen, die das System über das Segment wissen muss. Diese Information enthält: ist es gegenwärtig im Speicher; wenn im Speicher, wo ist es; Zugriffsrechte (z. B. read-only). Der

[4] Aus diesem Grund werden die Register (und ihre Inhalte) im protected Mode mit *Selektor*, im real Mode mit *Segment* bezeichnet. [Anm. d. Ü.]

Index des Eintrags für das Segment ist der Selektorwert, der im Selektorregister gespeichert ist.

Ein großer Nachteil des 16-bit protected Mode ist, dass Offsetwerte immer noch 16 bit Größen sind. Als Konsequenz daraus sind Segmentgrößen immer noch auf höchstens 64K beschränkt. Das macht die Benutzung großer Arrays problematisch!

Ein bekannter PC Kolumnist hat die 286 CPU „gehirntot“ genannt.

1.2.8 32-bit Protected Mode

Die 80386 führte den 32-bit protected Mode ein. Es gibt zwei hauptsächliche Unterschiede zwischen dem 386 32-bit und dem 286 16-bit protected Modus:

1. Offsets sind auf 32 bit erweitert. Das erlaubt Offsets im Bereich bis zu 4 Milliarden. Daher können Segmente Größen bis zu 4 Gigabyte haben.
2. Segmente können in kleinere, 4K große Einheiten unterteilt werden, die *Seiten* (*Pages*) genannt werden. Das virtuelle Speichersystem arbeitet nun mit Seiten anstatt Segmenten. Das bedeutet, dass zu jedem Zeitpunkt nur Teile eines Segments im Speicher sein müssen. Im 16-bit Modus der 286 ist entweder das ganze Segment im Speicher oder gar nichts davon. Das ist mit den großen Segmenten, die der 32-bit Modus ermöglicht, nicht praktikabel.

In Windows 3.x, bezieht sich *standard mode* auf den 16-bit protected Modus der 286 und *enhanced mode* bezieht sich auf den 32-bit Modus. Windows 9x, Windows NT/2000/XP, OS/2 und Linux laufen alle im paged 32-bit protected Mode.

1.2.9 Interrupts

Manchmal muss der gewöhnliche Programmfluss unterbrochen werden, um Ereignisse zu verarbeiten, die einer umgehenden Antwort bedürfen. Die Hardware eines Computers stellt einen Mechanismus, *Interrupts* genannt, bereit, um diese Ereignisse zu behandeln. Wenn zum Beispiel eine Maus bewegt wird, unterbricht die Hardware der Maus das laufende Programm um die Mausbewegung zu behandeln (um den Mauscursor zu bewegen, usw.). Interrupts bewirken, dass die Kontrolle an einen *Interrupt-Handler* übergeben wird. Interrupt-Handler sind Routinen, die Interruptanforderungen bedienen. Jeder Art von Interrupt ist eine ganze Zahl zugeordnet. Am Anfang des physikalischen Speichers liegt eine Tabelle von *Interrupt-Vektoren*, die die segmentierten Adressen der Interrupt-Handler enthält. Die Nummer der Interrupts ist im Wesentlichen ein Index in diese Tabelle.

Externe Interrupts haben ihren Ursprung außerhalb der CPU. (Die Maus ist ein Beispiel für diesen Typ.) Viele I/O Geräte generieren Interrupts (z. B. Tastatur, Zeitgeber, Laufwerke, CD-ROM und Soundkarten). Interne Interrupts haben ihren Ursprung innerhalb der CPU, entweder durch einen Fehler, oder durch den Interrupt-Befehl. Durch Fehler hervorgerufene Interrupts werden auch *Traps* genannt. Durch den Interrupt-Befehl generierte Interrupts werden *Software-Interrupts* genannt. DOS benutzt diese Interrupt-Typen um sein API (Application Programming Interface) zu implementieren. Moderne Betriebssysteme (wie Windows und UNIX) benutzen eine C-basierte Schnittstelle.[5]

[5] Jedoch können sie auf dem Kernel-Level eine Schnittstelle auf niederem Niveau benutzen.

Viele Interrupt-Handler geben die Kontrolle an das unterbrochene Programm zurück, wenn sie enden. Sie stellen alle Register wieder mit denselben Werten her, die sie hatten, bevor der Interrupt auftrat. Deshalb läuft das unterbrochene Programm weiter, als ob nichts geschehen wäre (außer, dass es einige CPU Zyklen verlor). Traps kehren gewöhnlich nicht zurück. Oft brechen sie das Programm ab.

1.3 Assemblersprache

1.3.1 Maschinensprache

Jeder CPU-Typ versteht seine eigene Maschinensprache. Befehle in Maschinensprache bestehen aus Zahlen, die als Bytes im Speicher abgelegt werden. Jeder Befehl hat seinen eigenen numerischen Code, der sein *Operations-Code* oder kurz *Opcode* genannt wird. Die Befehle der 8086 Prozessoren variieren in der Länge. Der Opcode befindet sich immer am Anfang des Befehls. Viele Befehle schließen auch Daten mit ein (z. B. Konstanten oder Adressen), die von dem Befehl benutzt werden.

Maschinensprache ist sehr schwierig, um direkt darin zu programmieren. Die Bedeutung der numerisch kodierten Befehle zu entziffern ist für Menschen mühsam. Zum Beispiel ist der Befehl, der sagt, die EAX und EBX Register zusammenzuzählen und das Ergebnis zurück nach EAX zu speichern, durch die folgenden hex-Codes verschlüsselt:

```
03 C3
```

Das ist schwerlich offensichtlich. Glücklicherweise kann ein Programm, *Assembler* genannt, diese mühselige Arbeit für den Programmierer tun.

1.3.2 Assemblersprache

Ein Programm in Assemblersprache wird als Text gespeichert (genauso wie ein Programm in einer Hochsprache). Jeder Assemblerbefehl entspricht genau einem Maschinenbefehl. Zum Beispiel würde der oben beschriebene Additions-Befehl in Assemblersprache so aussehen:

```
add eax, ebx
```

Hier ist die Bedeutung des Befehls *viel* klarer als in Maschinencode. Das Word `add` ist ein *Mnemonik* für den Additions-Befehl. Die allgemeine Form eines Assemblerbefehls ist:

```
Mnemonik Operand(en)
```

Es dauerte für Computerwissenschaftler mehrere Jahre, nur um herauszufinden, wie man überhaupt einen Compiler schreibt!

Ein *Assembler* ist ein Programm, das eine Textdatei mit Assemblerbefehlen liest und es in Maschinensprache umwandelt. *Compiler* sind Programme, die entsprechende Umwandlungen für Programme in Hochsprachen ausführen. Ein Assembler ist viel einfacher als ein Compiler. Jeder Befehl in Assemblersprache repräsentiert einen einzelnen Maschinenbefehl. Befehle in Hochsprachen sind sehr *viel* komplexer und können viele Maschinenbefehle erfordern.

Ein weiterer wichtiger Unterschied zwischen Assembler- und Hochsprache ist, da jeder unterschiedliche CPU-Typ seine eigene Maschinensprache hat, hat

er ebenso seine eigene Assemblersprache. Assemblerprogramme zwischen verschiedenen Computerarchitekturen zu portieren ist *sehr* viel schwieriger als in einer Hochsprache.

Die Beispiele in diesem Buch verwenden den Netwide Assembler oder kurz NASM. Er ist frei aus dem Internet erhältlich (siehe die URL im Vorwort). Weitere verbreitete Assembler sind Microsofts Assembler (MASM) oder Borlands Assembler (TASM). Es gibt einige Unterschiede in der Assembler-Syntax zwischen MASM/TASM und NASM.

1.3.3 Operanden der Befehle

Befehle in Maschinensprache haben unterschiedliche Anzahlen und Typen von Operanden; im Allgemeinen jedoch hat jeder Befehl selbst eine festgelegte Anzahl von Operanden (0 bis 3). Operanden können von folgendem Typ sein:

register: Diese Operanden beziehen sich direkt auf die Inhalte der Register der CPU.

memory: Diese beziehen sich auf Daten im Speicher. Die Adresse der Daten kann eine fest in den Befehl kodierte Konstante sein oder kann unter Benutzung von Werten in Registern berechnet werden. Adressen sind immer Offsets vom Anfang eines Segments.

immediate: Diese sind festgelegte Werte, die im Befehl selbst aufgeführt sind. Sie werden im Befehl selbst gespeichert (im Codesegment), nicht im Datensegment.

implied: Diese Operanden werden nicht explizit aufgeführt. Zum Beispiel addiert der Inkrement-Befehl eins zu einem Register oder Speicher. Die Eins ist implizit.

1.3.4 Grundlegende Befehle

Der grundlegendste Befehl ist der `MOV` Befehl. Er kopiert Daten von einem Ort an einen anderen (wie der Zuweisungsoperator in einer Hochsprache). Er benötigt zwei Operanden:

```
mov dest, src
```

Die durch *src* spezifizierten Daten werden nach *dest* kopiert. Eine Einschränkung ist, dass nicht beide Operanden Speicheroperanden sein können. Dies zeigt eine weitere Eigenart von Assembler auf. Es gibt öfters etwas willkürliche Regeln darüber, wie die verschiedenen Befehle benutzt werden können. Die Operanden müssen außerdem die gleiche Größe haben. Der Wert von AX kann nicht in BL gespeichert werden.

Hier ist ein Beispiel (Semikola beginnen einen Kommentar):

```
1 mov    eax, 3      ; speichere 3 ins EAX Register (3 ist immediate Operand)
2 mov    bx, ax      ; speichere den Wert von AX ins BX Register
```

Der Befehl `ADD` wird benutzt, um Ganzzahlen zu addieren.

```
3 add    eax, 4      ; eax = eax + 4
4 add    al, ah      ; al = al + ah
```

Der Befehl `SUB` subtrahiert Ganzzahlen.

```
5       sub     bx, 10      ; bx = bx - 10
6       sub     ebx, edi    ; ebx = ebx - edi
```

Die Befehle `INC` und `DEC` inkrementieren oder dekrementieren Werte um 1. Da der eine Operand impliziert ist, ist der Maschinencode für `INC` und `DEC` kürzer als für die entsprechenden `ADD` and `SUB` Befehle.

```
7       inc     ecx         ; ecx++
8       dec     dl          ; dl--
```

1.3.5 Direktiven

Eine *Direktive* ist ein Werkzeug des Assemblers und nicht der CPU. Sie werden im Allgemeinen benutzt, um entweder den Assembler zu etwas zu veranlassen oder ihn über etwas zu informieren. Sie werden nicht in Maschinencode übersetzt. Allgemein werden Direktiven benutzt um:

- Konstanten zu definieren
- Speicher zu definieren, in dem Daten gespeichert werden
- Speicher in Segmente zu gruppieren
- bedingten Quellcode einzuschließen
- andere Dateien einzuschließen

NASM Code wird durch einen Präprozessor geführt, genau wie in C. Er hat viele gleiche Präprozessor-Befehle wie C. Jedoch beginnen die Direktiven für den Präprozessor von NASM mit einem % anstatt mit einem # wie in C.

Die equ Direktive

Die Direktive `equ` kann verwendet werden um ein *Symbol* zu definieren. Symbole sind mit Namen versehene Konstanten, die in Assemblerprogrammen verwendet werden können. Das Format ist:

symbol `equ` *value*

Werte von Symbolen können später *nicht* neu definiert werden.

Die %define Direktive

Diese Direktive entspricht der `#define` Direktive von C. Im Allgemeinen wird sie meistens verwendet, um konstante Makros zu definieren, genauso wie in C.

```
%define SIZE 100
        mov     eax, SIZE
```

Der vorige Code definiert unter dem Namen `SIZE` ein Makro und zeigt seine Verwendung in einem `MOV` Befehl. Makros sind in zweierlei Hinsicht flexibler als Symbole. Makros können neu definiert werden und können mehr als einfache konstante Zahlen sein.

Daten Direktiven

Daten Direktiven werden in Datensegmenten verwendet, um Speicherplatz zu definieren. Es gibt zwei Methoden, mit denen Speicher reserviert werden kann. Die erste Methode definiert nur den Platz für die Daten; die zweite Methode definiert den Platz und einen Anfangswert. Die erste Methode benutzt eine der `RESx` Direktiven. Das *x* wird durch einen Buchstaben ersetzt, der die Größe des Objekts bestimmt, das gespeichert werden soll. Tabelle 1.3 zeigt die möglichen Werte.

Einheit	Buchstabe
byte	B
word	W
double word	D
quad word	Q
ten bytes	T

Tabelle 1.3: Buchstaben für `RESx` und `Dx` Direktiven

Die zweite Methode (die auch einen Startwert definiert) benutzt eine der `Dx` Direktiven. Die Buchstaben für *x* sind die gleichen wie die für die `RESx` Direktiven.

Es ist sehr verbreitet, Speicherstellen mit *Labels* zu markieren. Labels erlauben einem, sich im Code einfach auf Speicherstellen beziehen zu können. Unten sind verschiedene Beispiele:

```
1  L1   db    0              ; Byte namens L1, mit Anfangswert 0
2  L2   dw    1000           ; Wort namens L2, mit Anfangswert 1000
3  L3   db    110101b        ; Byte initialisiert mit binär 110101 (53 dezimal)
4  L4   db    12h            ; Byte initialisiert mit hex 12 (18 dezimal)
5  L5   db    17o            ; Byte initialisiert mit octal 17 (15 dezimal)
6  L6   dd    1A92h          ; Doppelwort initialisiert mit hex 1A92
7  L7   resb  1              ; 1 nicht-initialisiertes Byte
8  L8   db    "A"            ; Byte initialisiert mit ASCII Code für A (65)
```

Doppelte und einfache Anführungszeichen werden gleich behandelt. Nacheinander aufgeführte Datendefinitionen werden sequenziell im Speicher abgelegt. Das heißt, das Wort `L2` wird unmittelbar nach `L1` im Speicher gespeichert. Folgen von Speicheranforderungen können ebenso definiert werden.

```
9  L9   db    0, 1, 2, 3           ; definiert 4 Bytes
10 L10  db    "w", "o", "r", 'd', 0 ; definiert einen C String = "word"
11 L11  db    'word', 0            ; genau wie L10
```

Die `DD` Direktive kann benutzt werden, um sowohl Integer- als auch Fließpunkt-Konstanten[6] einfacher Genauigkeit zu definieren. Jedoch kann `DQ` nur verwendet werden, um Fließpunkt-Konstanten doppelter Genauigkeit zu definieren.

Für lange Folgen ist oft die `TIMES` Direktive von NASM nützlich. Diese Direktive wiederholt ihren Operanden eine gegebene Anzahl von Malen. Zum Beispiel:

[6] Fließpunkt-Werte einfacher Genauigkeit sind äquivalent zu einer `float` Variablen in C.

```
12  L12  times  100 db 0       ; äquivalent zu 100 (db 0)'s
13  L13  resw   100            ; reserviert Platz für 100 Words
```

Erinnern wir uns, dass Labels verwendet werden können, um im Code auf Daten zu verweisen. Es gibt zwei Wege, auf denen Labels verwendet werden können. Wenn ein Label selbst verwendet wird, wird es als die Adresse (oder der Offset) der Daten aufgefasst. Wird das Label in eckige Klammern gesetzt (`[]`), wird es als die Daten an dieser Adresse aufgefasst. In anderen Worten, man sollte das Label als einen *Zeiger* auf die Daten auffassen und die eckigen Klammern dereferenzieren den Zeiger genauso wie es der Asteriskus (`*`) in C macht. (MASM/TASM folgen einer anderen Konvention.) Im 32-bit Modus sind Adressen 32 bit groß. Hier sind einige Beispiele:

```
14        mov    al, [L1]       ; kopiere Byte von L1 in AL
15        mov    eax, L1        ; EAX = Adresse des Bytes bei L1
16        mov    [L1], ah       ; kopiere AH ins Byte bei L1
17        mov    eax, [L6]      ; kopiere Dword von L6 in EAX
18        add    eax, [L6]      ; EAX = EAX + Dword bei L6
19        add    [L6], eax      ; Dword bei L6 += EAX
20        mov    al, [L6]       ; kopiere erstes Byte vom Dword bei L6 in AL
```

Zeile 20 der Beispiele zeigt eine wichtige Eigenschaft von NASM. Der Assembler merkt sich *nicht* den Typ der Daten, auf den sich ein Label bezieht. Es liegt beim Programmierer, sicher zu stellen, dass er (oder sie) ein Label richtig verwendet. Später wird es geläufig werden, Adressen von Daten in Registern zu speichern und das Register wie einen Zeiger in C zu benutzen. Wiederum, es erfolgt keine Überprüfung, ob der Zeiger korrekt benutzt wird. Auf diese Weise ist Assembler sehr viel fehleranfälliger als sogar C.

Betrachten wir den folgenden Befehl:

```
21        mov    [L6], 1        ; speichere eine 1 nach L6
```

Dieses Statement ruft einen `operation size not specified` Fehler hervor. Warum? Weil der Assembler nicht weiß, ob er die 1 als ein Byte, Wort oder Doppelwort speichern soll. Um das zu korrigieren, fügt man eine Größenangabe hinzu:

```
22        mov    dword [L6], 1  ; speichere eine 1 nach L6
```

Dies sagt dem Assembler, dass er die 1 als ein Doppelwort, das bei `L6` beginnt, speichern soll. Andere Größenangaben sind: `BYTE`, `WORD`, `QWORD` und `TWORD`[7].

1.3.6 Eingabe und Ausgabe (I/O)

Eingabe und Ausgabe sind sehr systemabhängige Aktivitäten. Sie stehen mit den Schnittstellen zur Hardware des Systems in Verbindung. Hochsprachen wie C besitzen Routinen in Standard-Bibliotheken, die eine einfache, einheitliche Programmierschnittstelle für Ein- und Ausgabe zur Verfügung stellen. Assemblersprachen besitzen keine Standard-Bibliotheken. Sie müssen entweder direkt auf die Hardware zugreifen (was im protected Modus eine privilegierte Operation ist) oder benutzen was auch immer an low-level Routinen vom Betriebssystem bereitgestellt werden.

[7] `TWORD` definiert einen zehn Byte großen Bereich im Speicher. Die Fließpunkt-Einheit verwendet diesen Datentyp.

print_int	gibt auf dem Schirm den Wert des Integers in EAX aus
print_char	gibt auf dem Schirm das Zeichen aus, dessen ASCII Wert in AL gespeichert ist
print_string	gibt auf dem Schirm den Inhalt des Strings aus, dessen *Adresse* in EAX gespeichert ist. Der String muss ein C String (d. h. Null-terminiert) sein.
print_nl	gibt auf dem Schirm ein new-line Zeichen aus.
read_int	liest einen Integer von der Tastatur und speichert ihn im EAX Register.
read_char	liest ein einzelnes Zeichen von der Tastatur und speichert seinen ASCII Code im EAX Register.

Tabelle 1.4: Assembler I/O Routinen

Für Routinen in Assembler ist es sehr verbreitet, zusammen mit C verwendet zu werden. Ein Vorteil davon ist, dass der Assemblercode die I/O Routinen aus der Standard C Bibliothek verwenden kann. Jedoch muss man die Regeln kennen, mit denen Informationen zwischen Routinen, die C verwendet, ausgetauscht werden. Diese Regeln sind zu kompliziert, um hier behandelt zu werden. (Sie werden später betrachtet!) Um I/O zu vereinfachen, hat der Autor seine eigenen Routinen entwickelt, die die komplexen Regeln von C verstecken und eine sehr viel einfachere Schnittstelle bereitstellen. Tabelle 1.4 beschreibt die bereitgestellten Routinen. Alle Routinen erhalten die Werte in allen Registern, mit Ausnahme der Lese-Routinen. Diese Routinen verändern den Wert des EAX Registers. Um diese Routinen zu benutzen, muss man eine Datei mit Informationen einbinden, die der Assembler benötigt, um sie verwenden zu können. Um eine Datei in NASM einzubinden, benutzt man die `%include` Direktive des Präprozessors. Die folgende Zeile schließt die Datei, die von den I/O Routinen des Autors benötigt wird, mit ein:[8]

```
%include "asm_io.inc"
```

Um eine der Ausgaberoutinen zu verwenden, muss man EAX mit dem korrekten Wert laden und den `CALL` Befehl verwenden, um sie aufzurufen. Der `CALL` Befehl ist äquivalent zu einem Funktionsaufruf in einer Hochsprache. Zur Ausführung springt er zu einem anderen Abschnitt im Code, kehrt aber zu seinem Ursprung zurück, nachdem die Routine beendet ist. Das folgende Beispielprogramm (Seite 17) zeigt verschiedene Beispiele von Aufrufen dieser I/O-Routinen.

1.3.7 Debugging (Fehlersuche)

Die Bibliothek des Autors enthält auch einige nützliche Routinen, um Programme zu debuggen. Diese Debugroutinen stellen Informationen über den Zustand des Computers dar, ohne diesen Zustand zu verändern. Diese Routinen sind in Wirklichkeit *Makros*, die den gegenwärtigen Zustand der CPU festhalten

[8] Die `asm_io.inc` (und die `asm_io` Objektdatei, die `asm_io.inc` benötigt) sind in den Downloads der Beispielprogramme auf der Webseite für dieses Tutorial, `http://www.drpaulcarter.com/pcasm`, enthalten.

und dann eine Subroutine aufrufen. Die Makros sind in der oben erwähnten `asm_io.inc` Datei definiert. Makros werden wie gewöhnliche Befehle verwendet. Operanden von Makros werden durch Kommata getrennt.

Es gibt vier Debugroutinen mit Namen `dump_regs`, `dump_mem`, `dump_stack` und `dump_math`; sie zeigen jeweils die Werte der Register, von Speicher, Stack und mathematischem Coprozessor.

dump_regs Dieses Makro gibt die Werte der Register (in hexadezimal) des Computers über `stdout` (d. h. den Bildschirm) aus. Es zeigt ebenfalls die gesetzten Bits des FLAGS[9] Registers. Wenn zum Beispiel das Zero-Flag 1 ist, wird *ZF* ausgegeben. Ist es 0, wird nichts ausgegeben. Es hat ein einzelnes Integer-Argument, das ebenfalls ausgegeben wird. Dieses kann dazu benutzt werden, um die Ausgabe verschiedener `dump_regs` Befehle zu unterscheiden.

dump_mem Dieses Makro druckt den Inhalt eines Speicherbereichs (in hexadezimal) und ebenfalls als ASCII-Zeichen aus. Es verwendet drei, durch Kommata getrennte Parameter. Der erste ist ein Integer, der zur Markierung der Ausgabe verwendet wird (genauso wie das `dump_regs` Argument). Das zweite Argument ist die auszugebende Adresse. (Dies kann ein Label sein.) Das letzte Argument ist die Anzahl von 16-Byte Paragraphen, die ab dieser Adresse ausgegeben werden sollen. Der dargestellte Speicher beginnt an der ersten Paragraphengrenze vor der geforderten Adresse.

dump_stack Dieses Makro gibt die Werte auf dem Stack der CPU aus. (Der Stack wird in Kapitel 4 behandelt.) Der Stack ist in Doppelwörtern organisiert und diese Routine stellt sie so dar. Sie erwartet drei, durch Kommata getrennte Werte. Der erste ist eine Integer-Marke (wie bei `dump_regs`). Der zweite ist die Anzahl Doppelwörter, die *vor* der im `EBP`-Register enthaltenen Adresse ausgegeben werden und das dritte Argument ist die Anzahl der Doppelwörter, die *nach* der Adresse in `EBP` ausgegeben werden.

dump_math Dieses Makro druckt die Werte der Register des mathematischen Coprozessors aus. Es erwartet ein einzelnes Integer-Argument, das benutzt wird, um die Ausgabe zu markieren, genauso wie es das Argument von `dump_regs` tut.

1.4 Ein Programm erstellen

Heutzutage ist es ungewöhnlich, ein stand-alone Programm zu erstellen, das vollständig in Assembler geschrieben ist. Assembler wird gewöhnlich für bestimmte kritische Schlüsselroutinen verwendet. Warum? Es ist sehr *viel* einfacher in einer höheren Programmiersprache zu programmieren als in Assembler. Ebenso macht es die Benutzung von Assembler sehr schwierig, ein Programm auf andere Plattformen zu portieren. De facto ist es selten, überhaupt Assembler zu verwenden.

So, warum sollte überhaupt irgendjemand Assembler lernen?

1. Manchmal kann in Assembler geschriebener Code schneller und kleiner sein als durch Compiler generierter Code.

[9] Kapitel 2 behandelt dieses Register.

```
1  int main()
2  {
3    int ret_status ;
4    ret_status = asm_main();
5    return ret_status ;
6  }
```

Abbildung 1.6: `driver.c` Code

2. Assembler ermöglicht den Zugriff auf direkte Hardwareeigenschaften des Systems, die von einer Hochsprache aus schwierig oder nicht benutzt werden könnten.
3. In Assembler programmieren zu lernen hilft einem, ein tieferes Verständnis für die Arbeitsweise von Computern zu gewinnen.
4. In Assembler programmieren zu lernen hilft einem, besser zu verstehen, wie Compiler und Hochsprachen wie C arbeiten.

Die letzten beiden Punkte demonstrieren, dass das Lernen von Assembler hilfreich sein kann, selbst wenn man später nie darin programmiert. Tatsächlich programmiert der Autor selten in Assembler, aber er benutzt täglich die Ideen, die er daraus lernte.

1.4.1 Erstes Programm

Die ersten Programme in diesem Text werden alle von dem einfachen C Treiberprogramm in Abbildung 1.6 ausgehen. Es ruft einfach eine weitere Funktion namens `asm_main` auf. Das ist in Wirklichkeit die Routine, die in Assembler geschrieben wird. Es gibt verschiedene Vorteile, ein C Treiberprogramm zu benutzen. Erstens wird so das Programm durch das C System initialisiert, damit es korrekt im protected Mode läuft. Alle Segmente und ihre dazu gehörenden Segmentregister werden von C initialisiert. Der Assemblercode braucht sich darum überhaupt nicht zu kümmern. Zweitens wird dadurch auch die C Bibliothek für die Benutzung durch den Assemblercode verfügbar. Die I/O-Routinen des Autors ziehen Vorteil daraus. Sie benutzen die I/O Funktionen von C (`printf`, usw.). Nachfolgend ein einfaches Assemblerprogramm.

first.asm

```
1  ; Datei: first.asm
2  ; Erstes Assemblerprogramm. Dieses Programm fragt als
3  ; Eingabe nach zwei Integern und gibt ihre Summe aus.
4  ;
5  ; Um ein ausführbares Programm mit djgpp zu erzeugen:
6  ; nasm -f coff first.asm
7  ; gcc -o first first.o driver.c asm_io.o
8
9  %include "asm_io.inc"
10 ;
11 ; initialisierte Daten kommen in das .data Segment
12 ;
```

```
13  segment .data
14  ;
15  ; Diese Labels markieren Strings zur Ausgabe
16  ;
17  prompt1 db    "Enter a number: ", 0 ; Null nicht vergessen
18  prompt2 db    "Enter another number: ", 0
19  outmsg1 db    "You entered ", 0
20  outmsg2 db    " and ", 0
21  outmsg3 db    ", the sum of these is ", 0
22
23  ;
24  ; nicht-initialisierte Daten kommen in das .bss Segment
25  ;
26  segment .bss
27  ;
28  ; Diese Labels markieren die Dwords zur Speicherung der Eingabe
29  ;
30  input1  resd 1
31  input2  resd 1
32
33  ;
34  ; Code kommt in das .text Segment
35  ;
36  segment .text
37          global  _asm_main
38  _asm_main:
39          enter   0, 0              ; bereite Routine vor
40          pusha
41
42          mov     eax, prompt1      ; gib Prompt aus
43          call    print_string
44
45          call    read_int          ; lese Integer
46          mov     [input1], eax     ; in input1 speichern
47
48          mov     eax, prompt2      ; gib Prompt aus
49          call    print_string
50
51          call    read_int          ; lese Integer
52          mov     [input2], eax     ; in input2 speichern
53
54          mov     eax, [input1]     ; eax = Dword bei input1
55          add     eax, [input2]     ; eax += Dword bei input2
56          mov     ebx, eax          ; ebx = eax
57
58          dump_regs 1               ; gib Registerinhalte aus
59          dump_mem  2, outmsg1, 1   ; gib Speicher aus
60  ;
61  ; als nächstes gib Ergebnis Nachrichten in einer Folge von Schritten aus
62  ;
```

```
63          mov     eax, outmsg1
64          call    print_string      ; gib erste Nachricht aus
65          mov     eax, [input1]
66          call    print_int         ; gib input1 aus
67          mov     eax, outmsg2
68          call    print_string      ; gib zweite Nachricht aus
69          mov     eax, [input2]
70          call    print_int         ; gib input2 aus
71          mov     eax, outmsg3
72          call    print_string      ; gib dritte Nachricht aus
73          mov     eax, ebx
74          call    print_int         ; gib Summe (ebx) aus
75          call    print_nl          ; gib newline aus
76
77          popa
78          mov     eax, 0            ; zurück zu C
79          leave
80          ret
```

first.asm

Zeile 13 des Programms definiert ein Programmsegment, das Speicher spezifiziert, der im Datensegment (dessen Name `.data` ist) angelegt wird. Nur initialisierte Werte sollten in diesem Segment definiert werden. In den Zeilen 17 bis 21 werden einige Strings definiert. Sie werden durch die C Bibliothek ausgegeben und müssen deshalb mit einem *Null*-Zeichen (ASCII Code 0) beendet werden. Beachte, dass ein großer Unterschied zwischen `0` und `'0'` besteht.

Nicht-initialisierte Daten sollten im bss Segment (in Zeile 26 `.bss` genannt) deklariert werden. Der Namen dieses Segments rührt von einem anfänglichen UNIX-basierten Assembleroperator her, der „block started by symbol“ bedeutete. Ein Stacksegment gibt es ebenfalls. Es wird später besprochen werden.

Aus historischen Gründen wird das Codesegment `.text` genannt. Darin werden die Befehle abgelegt. Beachte, dass dem Code-Label für die Hauptroutine (Zeile 38) ein Unterstrich vorangestellt ist. Das ist Bestandteil der *Aufrufkonvention* von C. Diese Konvention spezifiziert die Regeln, die C benutzt, wenn es Code übersetzt. Es ist sehr wichtig, diese Konvention zu kennen, wenn C mit Assembler kombiniert werden soll. Später wird die gesamte Konvention präsentiert werden; für jetzt genügt es jedoch zu wissen, dass in C alle Symbole (d. h. Funktionen und globale Variable) vom C Compiler einen Unterstrich vorangestellt bekommen. (Diese Regel ist spezifisch für DOS/Windows, der Linux C Compiler stellt den C Symbolen überhaupt nichts voran.)

Die `global` Direktive in Zeile 37 teilt dem Assembler mit, dass er das `_asm_main` Symbol global machen soll. Im Gegensatz zu C haben Labels per Voreinstellung *internal scope*. Das bedeutet, dass nur Code im selben Modul das Label benutzen kann. Die `global` Direktive gibt den angegebenen Labels *external scope*. Auf diese Art von Label kann von jedem Modul im Programm aus zugegriffen werden. Das `asm_io` Modul erklärt die Labels `print_int`, et. al. als global. Das ist der Grund, warum man sie im `first.asm` Modul benutzen kann.

1.4.2 Compiler-Abhängigkeiten

Der obige Assemblercode ist spezifisch für den freien GNU[10]-basierten DJGPP C/C++ Compiler.[11] Dieser Compiler kann kostenlos aus dem Internet bezogen werden. Er benötigt einen 386-basierten PC oder besser und läuft unter DOS, Windows 95/98 oder NT. Dieser Compiler benutzt Objektdateien im COFF (Common Object File Format) Format. Um in dieses Format zu assemblieren, verwendet man mit `nasm` den `-f coff` Schalter (wie in den Kommentaren des obigen Codes gezeigt). Die Namenserweiterung der resultierenden Objektdatei wird `o` sein.

Der Linux C Compiler ist ebenfalls ein GNU Compiler. Um den obigen Code zu ändern, dass er unter Linux läuft, sind nur die Unterstrich-Präfixe in Zeilen 37 und 38 zu entfernen. Linux benutzt das ELF (Executable and Linkable Format) Format für die Objektdateien. Mit Linux benutzt man den `-f elf` Schalter. Er produziert ebenfalls eine Objektdatei mit einer `o` Namenserweiterung.

Die compilerspezifischen Beispieldateien, verfügbar von der Webseite des Autors, sind bereits abgeändert worden, um mit dem entsprechenden Compiler zu arbeiten.

Borland C/C++ ist ein weiterer populärer Compiler. Er benutzt das Microsoft OMF Format für Objektdateien. Mit Borlands C Compiler wird der `-f obj` Schalter verwendet. Die Namenserweiterung der Objektdatei wird `obj` sein. Das OMF-Format benutzt andere `segment` Direktiven als die anderen Objektformate. Das Datensegment (Zeile 13) muss geändert werden in:

```
segment _DATA public align=4 class=DATA use32
```

Das bss Segment (Zeile 26) muss geändert werden in:

```
segment _BSS public align=4 class=BSS use32
```

Das text Segment (Zeile 36) muss geändert werden in:

```
segment _TEXT public align=1 class=CODE use32
```

Zusätzlich sollte eine neue Zeile vor Zeile 36 eingefügt werden:

```
group DGROUP _BSS _DATA
```

Der C/C++ Compiler von Microsoft kann entweder das OMF oder das Win32 Format für Objektdateien benutzen. (Ein gegebenes OMF Format wird intern in das Win32 Format umgewandelt.) Das Win32 Format ermöglicht es, Segmente genauso wie für DJGPP und Linux zu definieren. Um in diesem Modus auszugeben, wird der `-f win32` Schalter benutzt. Die Namenserweiterung der Objektdatei wird `obj` sein.

1.4.3 Assemblierung des Codes

Der erste Schritt ist, den Code zu assemblieren. Von der Kommandozeile gibt man:

```
nasm -f object-format first.asm
```

ein, wobei *object-format*, abhängig davon, welcher C Compiler benutzt werden soll, entweder *coff*, *elf*, *obj* oder *win32* ist. (Erinnern wir uns daran, dass die Quelldatei sowohl für Linux als auch für Borland geändert werden muss.)

[10] GNU ist ein Projekt der Free Software Foundation (`http://www.fsf.org`)

[11] `http://www.delorie.com/djgpp`

1.4.4 Kompilation des C Codes

Kompilieren wir die `driver.c` Datei unter Verwendung eines C Compilers. Für DJGPP benutzt man:

```
gcc -c driver.c
```

Der `-c` Schalter meint, nur zu kompilieren, und noch nicht versuchen zu linken. Derselbe Schalter funktioniert genauso mit Linux, Borland und Microsoft Compilern.

1.4.5 Linken der Objektdateien

Linken ist der Prozess, der Maschinencode und Daten in Objektdateien und Bibliotheken zusammenzufügt, um ein ausführbares Programm zu schaffen. Wie weiter unten gezeigt wird, ist der Prozess kompliziert.

C Code erfordert die Standard C Bibliothek und speziellen *startup code* um zu laufen. Es ist *viel* einfacher, den Linker durch den C Compiler mit den korrekten Parametern aufrufen zu lassen, als zu versuchen, den Linker direkt aufzurufen. Um zum Beispiel den Code für das erste Programm mit DJGPP zu linken, verwendet man:

```
gcc -o first driver.o first.o asm_io.o
```

Das erzeugt ein ausführbares Programm mit Namen `first.exe` (oder nur `first` unter Linux).

Mit Borland würde man:

```
bcc32 first.obj driver.obj asm_io.obj
```

benutzen. Borland verwendet den Namen der ersten aufgeführten Datei, um den Namen der ausführbaren Datei festzulegen. So würde im obigen Fall das Programm `first.exe` genannt werden.

Es ist möglich, den Kompilier- und Linkschritt zu kombinieren. Zum Beispiel,

```
gcc -o first driver.c first.o asm_io.o
```

Damit wird `gcc driver.c` kompilieren und dann linken.

1.4.6 Die Ausgabe eines Assembler-Listings verstehen

Der Schalter `-l listing-file` kann verwendet werden, um `nasm` zu veranlassen, ein Listfile mit gegebenem Namen zu erzeugen. Diese Datei zeigt, wie der Code assembliert wurde. Hier ist aufgeführt, wie Zeilen 17 und 18 (des Datensegments) im Listfile erscheinen. (Die Zeilennummern stehen im Listfile; beachte aber, dass die Zeilennummern in den Quelldateien nicht die gleichen Zeilennummern sind wie die im Listfile.)

```
48 00000000 456E7465722061206E-     prompt1 db    "Enter a number: ", 0
49 00000009 756D6265723A2000
50 00000011 456E74657220616E6F-     prompt2 db    "Enter another number: ", 0
51 0000001A 74686572206E756D62-
52 00000023 65723A2000
```

Die erste Spalte jeder Zeile ist die Zeilennummer und die zweite ist der Offset (in hex) der Daten im Segment. Die dritte Spalte zeigt die rohen hex Werte, die gespeichert werden. In diesem Fall entsprechen die Hexdaten ASCII-Codes. Am Ende der Zeile ist dann der Text aus der Quelldatei eingefügt. Die in der zweiten Spalte aufgeführten Offsets sind sehr wahrscheinlich *nicht* die wahren Offsets, an denen die Daten im vollständigen Programm abgelegt werden. Jedes Modul kann seine eigenen Labels im Datensegment definieren (und auch in den anderen Segmenten). Im Linkschritt (siehe Abschnitt 1.4.5) werden alle diese Labeldefinitionen der Datensegmente zusammengefasst, um ein Datensegment zu bilden. Die neuen, endgültigen Offsets werden dann durch den Linker berechnet.

Hier ist ein kleiner Ausschnitt (Zeilen 54 bis 56 der Quellcodedatei) des Codesegments im Listfile:

```
94 0000002C A1[00000000]          mov     eax, [input1]
95 00000031 0305[04000000]        add     eax, [input2]
96 00000037 89C3                  mov     ebx, eax
```

Die dritte Spalte zeigt den durch den Assembler generierten Maschinencode. Oft kann jedoch der vollständige Code für eine Anweisung noch nicht berechnet werden. Zum Beispiel ist in Zeile 94 der Offset (oder Adresse) von `input1` nicht bekannt, bis der Code gelinkt wird. Der Assembler kann den Opcode für den `mov` Befehl berechnen (der nach dem Listing A1 ist), er schreibt aber den Offset in eckige Klammern, weil der genaue Wert noch nicht berechnet werden kann. In diesem Fall wird ein temporärer Offset von 0 benutzt, da `input1` am Anfang des Teils des bss Segments ist, der in dieser Datei definiert ist. Beachte, dass dies *nicht* bedeutet, dass es am Anfang des endgültigen bss Segments des Programms sein wird. Wird der Code gelinkt, setzt der Linker den korrekten Offset ein. Andere Befehle, wie Zeile 96, beziehen sich auf keine Labels. Hier kann der Assembler den vollständigen Maschinencode berechnen.

Big und little endian Darstellung

Wenn man sich Zeile 95 genauer ansieht, scheint etwas sehr seltsames mit dem Offset in eckigen Klammern des Maschinencodes zu sein. Das Label `input2` ist am Offset 4 (wie in dieser Datei definiert); jedoch ist der Offset, der im Speicher erscheint, nicht 00000004, sondern 04000000. Warum? Verschiedene Prozessoren speichern Multibyte-Integer in verschiedenen Bytefolgen im Speicher. Es gibt zwei gängige Methoden, um Integer zu speichern: *big endian* und *little endian*. Big endian ist die Methode, die am natürlichsten scheint. Das größte (d. h. höchstwertige Byte) wird zuerst gespeichert, dann das nächstgrößte, usw. Zum Beispiel würde das Doppelwort 00000004 als die 4 Bytes 00 00 00 04 gespeichert werden. IBM Mainframes, die meisten RISC Prozessoren und Prozessoren von Motorola verwenden alle diese big endian Methode. Jedoch verwenden Intelbasierte Prozessoren die little endian Methode! Hier wird das niederwertigste Byte zuerst gespeichert. So wird 00000004 als 04 00 00 00 im Speicher abgelegt. Dieses Format ist in der CPU fest verdrahtet und kann nicht geändert werden. Normalerweise braucht sich der Programmieren nicht darum zu kümmern, welches Format benutzt wird. Es gibt jedoch Umstände, unter denen es wichtig ist.

Endian wird wie Indien *ausgesprochen.*

1. Wenn binäre Daten zwischen verschiedenen Computern ausgetauscht werden (entweder durch Dateien oder über ein Netzwerk).

2. Wenn binäre Daten als Multibyte-Integer in den Speicher geschrieben werden und dann als individuelle Bytes zurückgelesen werden, oder umgekehrt.

Die Bytefolge wirkt sich nicht auf die Ordnung von Arrayelementen aus. Das erste Element eines Arrays ist immer an der niedersten Adresse. Das trifft auch auf Strings zu (die nur Character-Arrays sind). Die Bytefolge wirkt sich jedoch auf die einzelnen Elemente des Arrays aus.

1.5 Programmgerüst

Abbildung 1.7 zeigt eine Programmvorlage, die als Ausgangspunkt für die Entwicklung von Assemblerprogrammen dienen kann.

skel.asm

```
1  %include "asm_io.inc"
2  segment .data
3  ;
4  ; initialisierte Daten kommen hier ins Datensegment
5  ;
6
7  segment .bss
8  ;
9  ; nicht initialisierte Daten kommen ins bss Segment
10 ;
11
12 segment .text
13         global  _asm_main
14 _asm_main:
15         enter   0, 0              ; bereite Routine vor
16         pusha
17
18 ;
19 ; Code kommt in das text Segment. Nicht den Code vor
20 ; oder nach diesem Kommentar ändern.
21 ;
22
23         popa
24         mov     eax, 0            ; zurück zu C
25         leave
26         ret
```

skel.asm

Abbildung 1.7: Programmgerüst

Kapitel 2

Grundlagen der Assemblersprache

2.1 Arbeiten mit Integern (Ganzzahlen)

2.1.1 Die Darstellung von Integerwerten

Integer treten in zwei Geschmacksrichtungen auf: mit und ohne Vorzeichen. Vorzeichenlose Integer (die nicht-negativ sind) werden in einer nahe liegenden binären Weise repräsentiert. Die Zahl 200 als eine ein-Byte vorzeichenlose Ganzzahl würde als 11001000 (oder C8 in hex) repräsentiert werden.

Vorzeichenbehaftete Integer (die positiv oder negativ sein können) werden auf kompliziertere Weisen dargestellt. Betrachten wir zum Beispiel -56. $+56$ würde als Byte durch 00111000 dargestellt werden. Auf dem Papier könnte man -56 als -111000 repräsentieren, aber wie würde das in einem Byte im Computerspeicher repräsentiert werden? Wie würde das Minuszeichen gespeichert werden?

Es gibt drei allgemeine Techniken, die zur Darstellung von vorzeichenbehafteten Integern im Computerspeicher benutzt wurden. Alle diese Methoden benutzen das höchstwertige Bit des Integers als ein *Vorzeichenbit*. Dieses Bit ist 0, wenn die Zahl positiv ist und 1, wenn negativ.

Signed Magnitude

Die erste Methode ist die einfachste und wird *signed magnitude* genannt. Sie stellt den Integer in zwei Teilen dar. Der erste Teil ist das Vorzeichenbit und der zweite ist der Betrag des Integers. So würde 56 als das Byte $\underline{0}$0111000 (das Vorzeichenbit ist unterstrichen) dargestellt werden und -56 als $\underline{1}$0111000. Der größte Bytewert wird $\underline{0}$1111111 oder $+127$ sein und der kleinste Bytewert wäre $\underline{1}$1111111 oder -127. Um einen Wert zu negieren wird das Vorzeichenbit umgekehrt. Diese Methode ist einfach, hat aber ihre Nachteile. Zuerst gibt es zwei mögliche Werte für Null, $+0$ ($\underline{0}$0000000) und -0 ($\underline{1}$0000000). Da Null weder positiv noch negativ ist, sollten sich beide dieser Repräsentationen gleich verhalten. Das kompliziert die Logik für die Arithmetik der CPU. Zweitens ist die allgemeine Arithmetik ebenfalls kompliziert. Wenn 10 zu -56 addiert wird,

muss dies zu 10 subtrahiert von 56 umgedeutet werden. Wiederum kompliziert dies die Logik der CPU.

One's Complement (Einerkomplement)

Die zweite Methode ist als Repräsentation im *Einerkomplement* bekannt. Das Einerkomplement einer Zahl wird gefunden, indem jedes Bit in der Zahl invertiert wird. (Eine andere Betrachtungsweise besteht darin, den neuen Bitwert als $1 - \text{alterBitwert}$ anzusehen.) Das Einerkomplement von $\underline{0}0111000$ (+56) zum Beispiel ist $\underline{1}1000111$. In Einerkomplement-Notation ist das Berechnen des Einerkomplements gleichwertig zur Negation. Deshalb ist $\underline{1}1000111$ die Repräsentation von −56. Beachte, dass das Vorzeichenbit automatisch durch die Einerkomplementierung geändert wurde und dass, wie man auch erwarten würde, das Einerkomplement zwei Mal genommen, die ursprüngliche Zahl ergibt. Wie bei der ersten Methode gibt es zwei Repräsentationen der Null: $\underline{0}0000000$ (+0) und $\underline{1}1111111$ (−0). Arithmetik mit Einerkomplement-Zahlen ist kompliziert.

Es gibt einen nützlichen Trick, um das Einerkomplement einer Zahl in hexadezimal zu finden, ohne nach binär zu konvertieren. Der Trick besteht darin, die Hexziffern von F (oder 15 in dezimal) abzuziehen. Diese Methode nimmt an, dass die Anzahl Bits in der Zahl ein Vielfaches von 4 ist. Hier ist ein Beispiel: +56 ist 38 in hex. Um das Einerkomplement zu finden, zieht man jede Ziffer von F ab, um C7 in hex zu erhalten. Dies stimmt mit dem obigen Ergebnis überein.

Two's Complement (Zweierkomplement)

Die ersten beiden beschriebenen Methoden wurden auf frühen Computern benutzt. Moderne Computer benutzen eine dritte Methode, die *Zweierkomplement* genannt wird. Das Zweierkomplement einer Zahl wird durch die folgenden zwei Schritte gefunden:

1. Finde das Einerkomplement der Zahl
2. Addiere eins zum Ergebnis aus Schritt 1

Hier ist ein Beispiel unter Verwendung von $\underline{0}0111000$ (56). Zuerst wird das Einerkomplement berechnet: $\underline{1}1000111$. Dann wird eins addiert:

$$\begin{array}{rr} & \underline{1}1000111 \\ + & \qquad 1 \\ \hline & \underline{1}1001000 \end{array}$$

In Zweierkomplement-Darstellung ist die Berechnung des Zweierkomplements äquivalent zur Negation einer Zahl. So ist $\underline{1}1001000$ die Repräsentation von −56 im Zweierkomplement. Zwei Negationen sollten wieder die ursprüngliche Zahl geben. Überraschenderweise erfüllt das Zweierkomplement diese Forderung. Nimm das Zweierkomplement von $\underline{1}1001000$, indem eins zum Einerkomplement addiert wird.

$$\begin{array}{rr} & \underline{0}0110111 \\ + & \qquad 1 \\ \hline & \underline{0}0111000 \end{array}$$

Bei der Berechnung des Zweierkomplements kann die Addition der am weitesten links stehenden Bits einen Übertrag produzieren. Dieser Übertrag wird

nicht verwendet. Beachte, dass alle Daten im Computer eine feste Größe (in der Anzahl Bits) haben. Das Addieren zweier Bytes liefert immer ein Byte als Ergebnis (genauso wie die Addition zweier Wörter ein Wort liefert, usw.) Diese Eigenschaft ist wichtig für die Zweierkomplement-Notation. Betrachte zum Beispiel Null als eine ein-Byte Zweierkomplement-Zahl (10000000). Die Berechnung des Zweierkomplements liefert die Summe:

$$\begin{array}{rr} & \underline{1}1111111 \\ + & \qquad\quad 1 \\ \hline c & \underline{1}0000000 \end{array}$$

wobei c einen Übertrag repräsentiert. (Später wird gezeigt werden, wie dieser Übertrag entdeckt werden kann, er wird aber nicht im Ergebnis gespeichert.) So gibt es in der Zweierkomplement-Notation nur eine Null. Dies macht Arithmetik im Zweierkomplement einfacher als die vorheriger Methoden.

Bei Benutzung der Notation im Zweierkomplement kann ein vorzeichenbehaftetes Byte verwendet werden um die Zahlen -128 bis $+127$ zu repräsentieren. Tabelle 2.1 zeigt einige ausgewählte Werte. Werden 16 Bits verwendet, können die vorzeichenbehafteten Zahlen $-32\,768$ bis $+32\,767$ repräsentiert werden. $+32\,767$ wird dargestellt durch 7FFF, $-32\,768$ durch 8000, -128 als FF80 und -1 als FFFF. 32-bit Zweierkomplement-Zahlen reichen von ungefähr -2 Milliarden bis $+2$ Milliarden.

Zahl	Hex Repräsentation
0	00
1	01
127	7F
-128	80
-127	81
-2	FE
-1	FF

Tabelle 2.1: Darstellung im Zweierkomplement

Die CPU hat keine Vorstellung davon, was ein bestimmtes Byte (oder Wort oder Doppelwort) repräsentieren soll. Assembler hat nicht das Konzept von Datentypen, die eine Hochsprache hat. Wie Daten interpretiert werden, hängt davon ab, welche Befehle auf die Daten angewendet werden. Ob der Hexwert FF dazu bestimmt ist, eine vorzeichenbehaftete -1 oder eine vorzeichenlose $+255$ zu repräsentieren, hängt vom Programmierer ab. Die Sprache C definiert vorzeichenbehaftete und vorzeichenlose Integertypen. Diese ermöglicht dem C Compiler die richtigen Befehle zu bestimmen, um mit den Daten umzugehen.

2.1.2 Vorzeichenerweiterung

In Assembler haben alle Daten eine festgelegte Größe. Es ist nicht unüblich, die Größe der Daten ändern zu müssen, um sie mit anderen Daten zu benutzen. Die Größe zu verringern ist das Einfachste.

Einengung der Datengröße

Um die Größe der Daten zu verringern, entfernt man einfach die höherwertigen Bits der Daten. Hier ist ein triviales Beispiel:

```
1  mov    ax, 0034h        ; ax = 52 (in 16 Bits gespeichert)
2  mov    cl, al           ; cl = niedere 8 Bits von ax
```

Wenn die Zahl nicht korrekt in der kleineren Größe repräsentiert werden kann, schlägt die Herabsetzung der Größe natürlich fehl. Wenn zum Beispiel `AX` 0134h (oder 308 in dezimal) wäre, würde der obige Code `CL` immer noch auf 34h setzen. Diese Methode funktioniert sowohl mit vorzeichenbehafteten als auch mit vorzeichenlosen Zahlen. Betrachten wir vorzeichenbehaftete Zahlen. Wenn `AX` FFFFh (−1 als Wort) wäre, dann würde `CL` FFh (−1 als Byte) sein. Beachte jedoch, dass dies nicht korrekt ist, wenn der Wert in `AX` vorzeichenlos wäre!

Die Regel für vorzeichenlose Zahlen ist, dass alle entfernten Bits 0 sein müssen, damit die Konversion korrekt ist. Die Regel für vorzeichenbehaftete Zahlen ist, dass die entfernten Bits entweder alle 1 oder alle 0 sein müssen. Zusätzlich muss das erste nicht entfernte Bit denselben Wert haben wie die entfernten Bits. Dieses Bit wird zum neuen Vorzeichenbit des kleineren Wertes. Es ist wichtig, dass es gleich dem originalen Vorzeichenbit ist!

Ausweitung der Datengröße

Heraufsetzen der Größe der Daten ist komplizierter als herabsetzen. Betrachten wir das Hexbyte FF. Wenn es zu einem Wort erweitert wird, welchen Wert sollte dann das Wort haben? Es hängt davon ab, wie FF interpretiert wird. Ist FF ein vorzeichenloses Byte (255 in dezimal), dann sollte das Wort 00FF sein; wenn es jedoch ein vorzeichenbehaftetes Byte (−1 in dezimal) ist, dann sollte das Wort FFFF sein.

Um, ganz allgemein, eine vorzeichenlose Zahl zu erweitern, macht man alle neuen Bits der erweiterten Zahl zu 0. So wird FF zu 00FF. Um jedoch eine vorzeichenbehaftete Zahl zu erweitern, muss man das Vorzeichenbit *erweitern*. Das bedeutet, dass die neuen Bits Kopien des Vorzeichenbits werden. Da das Vorzeichenbit von FF 1 ist, müssen die neuen Bits ebenso alle Einsen sein, um dann FFFF zu liefern. Wenn die vorzeichenbehaftete Zahl 5A (90 in dezimal) erweitert wird, würde das Ergebnis 005A sein.

Es gibt mehrere Befehle, die die 80386 für die Zahlenerweiterung bereitstellt. Erinnern wir uns, dass der Computer nicht weiß, ob eine Zahl vorzeichenbehaftet oder vorzeichenlos ist. Es liegt am Programmierer, den richtigen Befehl zu verwenden.

Für vorzeichenlose Zahlen kann man mit einem `MOV` Befehl einfach Nullen in die oberen Bits laden. Um zum Beispiel das Byte in AL zu einem vorzeichenlosen Word in AX zu erweitern:

```
3  mov    ah, 0            ; setze obere 8 Bits auf Null
```

Jedoch ist es nicht möglich, einen `MOV` Befehl zu verwenden, um das vorzeichenlose Wort in AX zu einem vorzeichenlosen Doppelwort in EAX zu konvertieren. Warum nicht? Es gibt keinen Weg, um mit einem `MOV` die oberen 16 Bits von

EAX zu spezifizieren. Die 80386 löst dieses Problem, indem sie die neue Instruktion `MOVZX` bereitstellt. Dieser Befehl hat zwei Operanden. Die Datensenke (erster Operand) muss ein 16 oder 32 bit Register sein. Die Quelle (zweiter Operand) kann ein 8 oder 16 bit Register oder ein Byte oder Wort im Speicher sein. Die andere Einschränkung ist, dass die Senke größer als die Quelle sein muss. (Die meisten Befehle erfordern, dass Quelle und Ziel von der gleichen Größe sind.) Hier sind einige Beispiele:

```
4  movzx  eax, ax          ; erweitert ax zu eax
5  movzx  eax, al          ; erweitert al zu eax
6  movzx  ax, al           ; erweitert al zu ax
7  movzx  ebx, ax          ; erweitert ax zu ebx
```

Für vorzeichenbehaftete Zahlen gibt es keinen einfachen Weg, um den `MOV` Befehl in jedem Fall zu benutzen. Die 8086 lieferte mehrere Befehle, um vorzeichenbehaftete Zahlen zu erweitern. Der `CBW` (Convert Byte to Word) Befehl führt die Vorzeichenerweiterung des AL Registers nach AX durch. Die Operanden sind implizit. Der `CWD` (Convert Word to Double word) Befehl erweitert das Vorzeichen in AX nach DX:AX. Die Notation DX:AX bedeutet, die DX und AX Register als ein 32 bit Register aufzufassen, mit den oberen 16 Bits in DX und den unteren Bits in AX. (Erinnern wir uns daran, dass die 8086 kein 32 bit Register hat!) Die 80386 fügte mehrere neue Befehle hinzu. Der `CWDE` (Convert Word to Double word Extended) Befehl erweitert das Vorzeichen von AX nach EAX. Der `CDQ` (Convert Double word to Quad word) Befehl erweitert das Vorzeichen von EAX nach EDX:EAX (64 Bit!). Schließlich arbeitet der `MOVSX` Befehl wie `MOVZX`, außer dass er die Regeln für vorzeichenbehaftete Zahlen benutzt.

Anwendung in der C Programmierung

Die Erweiterung vorzeichenloser und vorzeichenbehafteter Integer tritt auch in C auf. Variable in C können entweder als vorzeichenbehaftet (signed) oder vorzeichenlos (unsigned) deklariert werden (`int` ist mit Vorzeichen). Betrachten wir den Code in Abbildung 2.1. In Zeile 3 wird die Variable `a` unter Verwendung der Regeln für vorzeichenlose Werte erweitert (unter Benutzung von `MOVZX`), aber in Zeile 4 werden die vorzeichenbehafteten Regeln für `b` benutzt (unter Benutzung von `MOVSX`).

ANSI C definiert nicht, ob der Typ `char` *mit oder ohne Vorzeichen ist, es liegt an jedem individuellen Compiler, das zu entscheiden. Deshalb wird der Typ in Abbildung 2.1 explizit definiert.*

```
1  unsigned char uchar = 0xFF;
2  signed char   schar = 0xFF;
3  int a = (int) uchar;      /* a = 255 (0x000000FF) */
4  int b = (int) schar;      /* b = -1 (0xFFFFFFFF) */
```

Abbildung 2.1: Ausweitung von `char` Werten

Es gibt einen verbreiteten Programmierfehler in C, der direkt mit diesem Thema in Verbindung steht. Betrachten wir den Code in Abbildung 2.2. Der Prototyp von `fgetc()` ist:

```
int fgetc( FILE * );
```

Man könnte sich fragen, warum die Funktion einen `int` zurückgibt, wenn sie doch Zeichen liest? Der Grund liegt darin, dass sie normalerweise einen `char`

(unter Verwendung der Null-Erweiterung zu einem `int` erweitert) zurückgibt. Jedoch gibt es einen Wert, den sie zurückgeben kann, der kein Zeichen ist, nämlich `EOF`. Das ist ein Makro, das gewöhnlich als -1 definiert ist. Folglich gibt `fgetc()` entweder ein zu einem `int` erweiterten `char` Wert (das in hex `000000`*`xx`* wäre) oder `EOF` (das in hex wie `FFFFFFFF` aussieht) zurück.

```
1  char ch;
2  while( (ch = fgetc(fp)) != EOF ) {
3    /* mache etwas mit ch */
4  }
```

Abbildung 2.2: I/O Fehler

Das grundlegende Problem mit dem Programm in Abbildung 2.2 ist, dass `fgetc()` einen `int` zurückgibt, der Wert aber in einem `char` gespeichert wird. C wird die höherwertigen Bits abschneiden, um den `int` Wert in einen `char` zu pressen. Das einzige Problem ist, dass die Zahlen (in hex) `000000FF` und `FFFFFFFF` beide zum Byte `FF` verkleinert werden. Deshalb kann die while-Schleife nicht zwischen dem von der Datei gelesenen Byte `FF` und dem Dateiende unterscheiden.

Was der Code in diesem Fall genau tut, hängt davon ab, ob `char` mit oder ohne Vorzeichen ist. Warum? Weil in Zeile 2 `ch` mit `EOF` verglichen wird. Da `EOF` ein `int` Wert ist,[1] wird `ch` zu einem `int` erweitert, sodass die beiden verglichenen Werte von der gleichen Größe sind.[2] Wie Abbildung 2.1 zeigte, ist es sehr wichtig, ob die Variable mit oder ohne Vorzeichen ist.

Ist `char` ohne Vorzeichen, wird `FF` zu `000000FF` erweitert. Dies wird mit `EOF` (`FFFFFFFF`) verglichen und als nicht gleich gefunden. Deshalb wird die Schleife niemals beendet!

Ist `char` mit Vorzeichen, wird `FF` zu `FFFFFFFF` erweitert. Der Vergleich wird wahr und die Schleife endet. Da das Byte `FF` jedoch von der Datei gelesen werden kann, könnte die Schleife vorzeitig beendet werden.

Die Lösung dieses Problems ist, die Variable `ch` als einen `int`, nicht als `char` zu definieren. Wird dies getan, wird in Zeile 2 weder abgeschnitten noch erweitert. Innerhalb der Schleife ist es sicher, den Wert abzuschneiden, da `ch` dort wirklich ein einfaches Byte sein *muss*.

2.1.3 Arithmetik im Zweierkomplement

Wie früher gezeigt wurde, führt der `add` Befehl Additionen und der `sub` Befehl führt Subtraktionen durch. Zwei der Bits im FLAGS Register, die diese Befehle setzen, sind das *Overflow* und das *Carry Flag*. Das Overflowflag wird gesetzt, wenn das wahre Ergebnis der Operation zu groß ist, um bei vorzeichenbehafteter Arithmetik in das Ziel zu passen. Das Carryflag wird gesetzt, wenn es einen Übertrag im MSB einer Addition oder einer Subtraktion gibt. Deshalb kann es verwendet werden, um einen Übertrag bei vorzeichenloser Arithmetik zu entdecken. Der Gebrauch des Carryflags für vorzeichenbehaftete Arithmetik wird

[1] Es ist ein allgemeines Missverständnis, dass Dateien ein EOF Zeichen an ihrem Ende hätten. Dies ist *nicht* der Fall!

[2] Der Grund für diese Forderung wird später gezeigt werden.

in Kürze gezeigt werden. Einer der großen Vorteile des 2er Komplements ist, dass die Regeln für Addition und Subtraktion genau die gleichen sind wie für vorzeichenlose Integer. Deshalb können `add` und `sub` für Integer mit und ohne Vorzeichen verwendet werden.

$$\begin{array}{rr} 002C & 44 \\ + \quad FFFF & + \quad (-1) \\ \hline 002B & 43 \end{array}$$

Dabei wird ein Übertrag gebildet, der aber nicht Bestandteil der Antwort ist.

Es gibt zwei verschiedene Multiplizier- und Divisionsbefehle. Um zu multiplizieren, verwendet man entweder den `MUL` oder den `IMUL` Befehl. Der `MUL` Befehl wird benutzt, um vorzeichenlose Integer zu multiplizieren und `IMUL` wird benutzt, um vorzeichenbehaftete Integer zu multiplizieren. Warum werden zwei verschiedene Befehle benötigt? Die Regeln für die Multiplikation sind für vorzeichenlose und vorzeichenbehaftete Zahlen im 2er Komplement unterschiedlich. Wie kommt das? Betrachten wir die Multiplikation des Bytes FF mit sich selbst zu einem Ergebnis mit Wortgröße. Unter Benutzung von vorzeichenloser Multiplikation ist dies 255 mal 255 oder 65 025 (oder FE01 in hex). Mit vorzeichenbehafteter Multiplikation ist dies -1 mal -1 oder 1 (0001 in hex).

Es gibt verschiedene Formen der Multiplikationsbefehle. Die älteste Form sieht so aus:

```
mul   source
```

source ist entweder ein Register oder eine Speicherreferenz. Es kann kein unmittelbarer Wert sein. Welche Multiplikation genau ausgeführt wird, hängt von der Größe des Quelloperanden ab. Ist der Operand von Bytegröße, wird er mit dem Byte im AL Register multipliziert und das Ergebnis wird in den 16 Bits von AX gespeichert. Hat die Quelle 16 Bits, wird sie mit dem Wort in AX multipliziert und das 32 bit Ergebnis wird in DX:AX gespeichert. Hat die Quelle 32 Bits, wird sie mit EAX multipliziert und das 64 bit Ergebnis wird nach EDX:EAX gespeichert.

dest	**source1**	**source2**	**Aktion**
	reg/mem8		AX = AL $\star$ source1
	reg/mem16		DX:AX = AX $\star$ source1
	reg/mem32		EDX:EAX = EAX $\star$ source1
reg16	reg/mem16		dest $\star$= source1
reg32	reg/mem32		dest $\star$= source1
reg16	immed8		dest $\star$= immed8
reg32	immed8		dest $\star$= immed8
reg16	immed16		dest $\star$= immed16
reg32	immed32		dest $\star$= immed32
reg16	reg/mem16	immed8	dest = source1 $\star$ source2
reg32	reg/mem32	immed8	dest = source1 $\star$ source2
reg16	reg/mem16	immed16	dest = source1 $\star$ source2
reg32	reg/mem32	immed32	dest = source1 $\star$ source2

Tabelle 2.2: `imul` Befehle

Der `IMUL` Befehl hat die gleichen Formate wie `MUL`, fügt aber einige weitere Befehlsformen hinzu. Es gibt Formate mit zwei und drei Operanden:

```
imul   dest, source1
imul   dest, source1, source2
```

Tabelle 2.2 zeigt die möglichen Kombinationen.

Die zwei Divisionsbefehle sind `DIV` und `IDIV`. Sie führen Integerdivisionen ohne bzw. mit Vorzeichen aus. Das allgemeine Format ist:

```
div   source
```

Wenn die Quelle 8 bit groß ist, dann wird AX durch den Operanden geteilt. Der Quotient wird in AL gespeichert und der Rest in AH. Hat die Quelle 16 Bits, dann wird DX:AX durch den Operanden dividiert. Der Quotient wird in AX gespeichert, der Rest in DX. Hat die Quelle 32 Bits, wird EDX:EAX durch den Operanden geteilt, der Quotient in EAX gespeichert und der Rest in EDX. Der `IDIV` Befehl arbeitet auf die gleiche Weise. Es gibt keine speziellen `IDIV` Befehlsformen wie bei `IMUL`. Wenn der Quotient zu groß ist um in sein Register zu passen oder der Teiler Null ist, wird das Programm unterbrochen und beendet. Ein sehr verbreiteter Fehler ist es, vor der Division zu vergessen DX oder EDX zu initialisieren.

Der `NEG` Befehl negiert seinen einzigen Operanden, indem er dessen Zweierkomplement berechnet. Sein Operand kann jedes 8-, 16- oder 32-bit Register oder Speicherstelle sein.

2.1.4 Beispielprogramm

math.asm

```
1  %include "asm_io.inc"
2  segment .data                    ; Ausgabe-Strings
3  prompt          db    "Enter a number: ", 0
4  square_msg      db    "Square of input is ", 0
5  cube_msg        db    "Cube of input is ", 0
6  cube25_msg      db    "Cube of input times 25 is ", 0
7  quot_msg        db    "Quotient of cube/100 is ", 0
8  rem_msg         db    "Remainder of cube/100 is ", 0
9  neg_msg         db    "The negation of the remainder is ", 0
10
11 segment .bss
12 input   resd    1
13
14 segment .text
15         global  _asm_main
16 _asm_main:
17         enter   0, 0             ; bereite Routine vor
18         pusha
19
20         mov     eax, prompt
21         call    print_string
22
23         call    read_int
24         mov     [input], eax
25
26         imul    eax              ; edx:eax = eax * eax
27         mov     ebx, eax         ; sichere Antwort in ebx
28         mov     eax, square_msg
```

```
29          call    print_string
30          mov     eax, ebx
31          call    print_int
32          call    print_nl
33
34          mov     ebx, eax
35          imul    ebx, [input]      ; ebx *= [input]
36          mov     eax, cube_msg
37          call    print_string
38          mov     eax, ebx
39          call    print_int
40          call    print_nl
41
42          imul    ecx, ebx, 25      ; ecx = ebx*25
43          mov     eax, cube25_msg
44          call    print_string
45          mov     eax, ecx
46          call    print_int
47          call    print_nl
48
49          mov     eax, ebx
50          cdq                       ; initialisiere edx durch Vorzeichenerweiterung
51          mov     ecx, 100          ; kann nicht durch unmittelbaren Wert teilen
52          idiv    ecx               ; edx:eax / ecx
53          mov     ecx, eax          ; sichere Quotient in ecx
54          mov     eax, quot_msg
55          call    print_string
56          mov     eax, ecx
57          call    print_int
58          call    print_nl
59          mov     eax, rem_msg
60          call    print_string
61          mov     eax, edx
62          call    print_int
63          call    print_nl
64
65          neg     edx               ; negiere den Teilerrest
66          mov     eax, neg_msg
67          call    print_string
68          mov     eax, edx
69          call    print_int
70          call    print_nl
71
72          popa
73          mov     eax, 0            ; kehre zu C zurück
74          leave
75          ret
```

math.asm

2.1.5 Arithmetik mit erhöhter Genauigkeit

Die Assemblersprache besitzt ebenso Befehle, die einem erlauben, Addition und Subtraktion auch mit Zahlen durchzuführen, die größer als Doppelwörter sind. Diese Befehle benutzen das Carryflag. Wie oben erwähnt, modifizieren `ADD` und `SUB` Befehle das Carryflag, wenn ein Übertrag generiert wird. Diese im Carryflag gespeicherte Information kann benutzt werden, um große Zahlen zu addieren oder subtrahieren, indem die Operation in einzelne Doppelwort- (oder kleinere) Stücke aufgeteilt wird.

Die `ADC` und `SBB` Befehle benutzen diese Information im Carryflag. Der `ADC` Befehl führt die folgende Operation durch:

operand1 = *operand1* + carry flag + *operand2*

Der `SBB` Befehl führt aus:

operand1 = *operand1* - carry flag - *operand2*

Wie werden diese benutzt? Betrachten wir die Summe von 64 bit Integern in EDX:EAX und EBC:ECX. Der folgende Code würde die Summe in EDX:EAX speichern:

```
1 add    eax, ecx          ; addiere untere 32 Bits
2 adc    edx, ebx          ; addiere obere 32 Bits und Übertrag
```

Die Subtraktion ist sehr ähnlich. Folgender Code zieht EBX:ECX von EDX:EAX ab:

```
3 sub    eax, ecx          ; subtrahiere untere 32 Bits
4 sbb    edx, ebx          ; subtrahiere obere 32 Bits und Übertrag
```

Für *wirklich* große Zahlen könnte eine Schleife benutzt werden (siehe Abschnitt 2.2). In einer Summationsschleife würde es bequemer sein, den `ADC` Befehl bei jeder Iteration zu verwenden (anstatt für alle außer der ersten Iteration). Das kann getan werden, wenn der `CLC` (CLear Carry) Befehl direkt vor der Schleife verwendet wird, um das Carryflag mit 0 zu initialisieren. Wenn das Carryflag 0 ist, gibt es keine Unterschiede zwischen den `ADD` und `ADC` Befehlen. Die gleiche Idee kann auch für die Subtraktion verwendet werden.

2.2 Kontrollstrukturen

Hochsprachen verfügen über Kontrollstrukturen auf einem hohen Niveau (z. B. die *if* und *while* Statements), die den Ausführungsfluss kontrollieren. Assembler bietet keine solchen komplexen Kontrollstrukturen. Er benutzt stattdessen das berüchtigte *goto* und unangemessen benutzt, kann es zu Spaghetticode führen! Es *ist* jedoch möglich, strukturierte Assemblerprogramme zu schreiben. Die grundsätzliche Vorgehensweise ist, die Programme möglichst unter Verwendung der vertrauten Kontrollstrukturen der Hochsprachen zu entwerfen und den Entwurf in die entsprechende Assemblersprache zu übersetzen (etwa so, wie es ein Compiler machen würde).

2.2.1 Vergleiche

Kontrollstrukturen entscheiden auf der Grundlage des Vergleichs von Daten, was zu tun ist. In Assembler wird das Ergebnis eines Vergleichs im FLAGS Register (Tabelle 2.3) gespeichert, um später benutzt zu werden. Die 80x86 stellt den `CMP` Befehl zur Verfügung, um Vergleiche durchzuführen. Das FLAGS Register wird auf der Grundlage der Differenz der beiden Operanden des `CMP` Befehls gesetzt. Die Operanden werden subtrahiert und die FLAGS werden auf Grund des Ergebnisses gesetzt, allerdings wird das Ergebnis *nirgends* gespeichert. Wenn man das Ergebnis benötigt, benutzt man den `SUB` anstatt des `CMP` Befehls.

Bit	7	6	5	4	3	2	1	0
Flag	SF	ZF	0	AF	0	PF	1	CF
	sign	zero		aux		parity		carry

Tabelle 2.3: Die Flagbits im unteren Byte des (E)FLAGS Registers

Für vorzeichenlose Integer sind zwei Flags (Bits im FLAGS Register) wichtig: das Zero- (ZF) und das Carry-Flag (CF). Das Zeroflag wird gesetzt (1), wenn die resultierende Differenz Null sein würde. Das Carryflag wird als Borrowflag bei der Subtraktion benutzt. Betrachten wir einen Vergleich wie:

```
cmp    vleft, vright
```

Die Differenz `vleft - vright` wird berechnet und die Flags entsprechend gesetzt. Ist die Differenz von `CMP` Null, `vleft = vright`, dann wird ZF gesetzt (d. h. 1) und CF gelöscht (d. h. 0). Ist `vleft > vright`, dann wird ZF gelöscht und CF wird gelöscht (kein Borrow). Ist `vleft < vright`, dann wird ZF gelöscht und CF wird gesetzt (Borrow).

Für Integer mit Vorzeichen gibt es drei Flags, die wichtig sind: das Zeroflag (ZF), das Overflowflag (OF) und das Signflag (SF). Das Overflowflag wird gesetzt, wenn das Ergebnis einer Operation überläuft (oder unterläuft). Das Signflag wird gesetzt, wenn das Ergebnis einer Operation negativ ist. Ist `vleft = vright`, wird das ZF gesetzt (genauso wie für vorzeichenlose Integer). Ist `vleft > vright`, wird ZF gelöscht und SF = OF. Ist `vleft < vright`, wird ZF gelöscht und SF $\neq$ OF.

Warum ist SF = OF, wenn `vleft > vright`? Wenn es keinen Überlauf gibt, dann hat die Differenz den richtigen Wert und muss nicht-negativ sein. Deshalb ist SF = OF = 0. Jedoch, wenn es einen Überlauf gibt, wird die Differenz nicht den richtigen Wert haben (und wird tatsächlich negativ sein). Folglich ist SF = OF = 1.

Vergessen Sie nicht, dass auch andere Befehle das FLAGS Register ändern können, nicht nur `CMP`.

2.2.2 Sprungbefehle

Sprungbefehle können die Ausführung zu beliebigen Punkten eines Programms führen. In anderen Worten, sie wirken wie ein *Goto*. Es gibt zwei Arten von Sprungbefehlen: unbedingte und bedingte. Ein unbedingter Sprung ist genau wie ein Goto, die Verzweigung wird immer durchgeführt. Ein bedingter Sprung kann die Verzweigung durchführen oder nicht, abhängig von den Flags im FLAGS Register. Führt ein bedingter Sprung die Verzweigung nicht durch, geht die Kontrolle zum nächsten Befehl über.

Der `JMP` (kurz für *jump*) Befehl führt unbedingte Sprünge aus. Sein einziges Argument ist gewöhnlich ein *Codelabel* des Befehls, zu dem gesprungen werden soll. Der Assembler oder Linker wird das Label durch die korrekte Adresse

des Befehls ersetzen. Dies ist eine weitere der mühseligen Operationen, die der Assembler ausführt, um das Leben des Programmierers einfacher zu machen. Es ist wichtig, sich zu vergegenwärtigen, dass der Befehl unmittelbar nach dem `JMP` Befehle niemals ausgeführt wird, es sei denn, ein anderer Befehl verzweigt zu ihm!

Es gibt verschiedene Varianten des Sprungbefehls:

SHORT Dieser Sprung ist in der Reichweite sehr begrenzt. Er kann nur um 128 Bytes im Speicher vor oder zurück springen. Der Vorteil dieses Typs ist, dass er weniger Speicher als die anderen benötigt. Er verwendet ein einzelnes vorzeichenbehaftetes Byte um das *Displacement* des Sprungs zu speichern. Der Wert des Displacements entscheidet, um wie viele Bytes vor oder zurück gesprungen werden soll. (Das Displacement wird zu EIP addiert.) Um einen kurzen Sprung zu spezifizieren, benutzt man das Schlüsselwort `SHORT` unmittelbar vor dem Label im `JMP` Befehl.

NEAR Dieser Sprung ist der vorgegebene Typ sowohl für unbedingte als auch für bedingte Sprünge; er kann verwendet werden, um zu jeder Stelle in einem Segment zu springen. Tatsächlich unterstützt die 80386 zwei Typen von nahen Sprüngen. Einer verwendet zwei Bytes für das Displacement. Dies erlaubt einem, sich ungefähr 32 000 Bytes vor oder zurück zu bewegen. Der andere Typ benutzt vier Bytes für das Displacement, das einem natürlich ermöglicht, sich zu jeder Stelle im Codesegment zu bewegen. Der Typ mit vier Bytes ist der vorgegebene im protected Mode der 386. Der Typ mit zwei Bytes kann spezifiziert werden, indem das Schlüsselwort `WORD` vor das Label im `JMP` Befehl gestellt wird.

FAR Dieser Sprung erlaubt der Kontrolle, sich in ein anderes Codesegment zu bewegen. Dies zu tun ist im protected Mode der 386 eine sehr seltene Sache.

Gültige Codelabels folgen denselben Regeln wie Datenlabels. Codelabels werden definiert, indem sie im Codesegment vor die Anweisung, die sie markieren, gesetzt werden. An das Label wird am Ort seiner Definition ein Doppelpunkt angehängt. Der Doppelpunkt ist *nicht* Bestandteil des Namens.

JZ	verzweigt nur, wenn ZF gesetzt ist
JNZ	verzweigt nur, wenn ZF nicht gesetzt ist
JO	verzweigt nur, wenn OF gesetzt ist
JNO	verzweigt nur, wenn OF nicht gesetzt ist
JS	verzweigt nur, wenn SF gesetzt ist
JNS	verzweigt nur, wenn SF nicht gesetzt ist
JC	verzweigt nur, wenn CF gesetzt ist
JNC	verzweigt nur, wenn CF nicht gesetzt ist
JP	verzweigt nur, wenn PF gesetzt ist
JNP	verzweigt nur, wenn PF nicht gesetzt ist

Tabelle 2.4: Einfache bedingte Verzweigungen

Es gibt viele verschiedene bedingte Sprunganweisungen. Auch sie benötigen ein Codelabel als ihren einzigen Operanden. Die einfachsten betrachten nur ein einziges Flag im FLAGS Register, um zu entscheiden, ob sie verzweigen oder nicht. Siehe Tabelle 2.4 für eine Liste dieser Instruktionen. (PF ist das

Parityflag, das anzeigt, ob die Anzahl der gesetzten Bits in den niederwertigen 8 bit des Ergebnisses gerade oder ungerade ist.)

Der folgende Pseudocode:

```
if ( EAX == 0 )
  EBX = 1;
else
  EBX = 2;
```

könnte in Assembler geschrieben werden als:

```
1       cmp    eax, 0           ; setze Flags (ZF gesetzt, wenn eax - 0 = 0)
2       jz     thenblock        ; wenn ZF gesetzt ist verzweige zu thenblock
3       mov    ebx, 2           ; ELSE Teil von IF
4       jmp    next             ; überspringe THEN Teil von IF
5   thenblock:
6       mov    ebx, 1           ; THEN Teil von IF
7   next:
```

Andere Vergleiche sind unter Verwendung der bedingten Verzweigungen in Tabelle 2.4 nicht so einfach. Um das zu zeigen, betrachten wie den folgenden Pseudocode:

```
if ( EAX >= 5 )
  EBX = 1;
else
  EBX = 2;
```

Wenn EAX größer als oder gleich fünf ist, dann kann das ZF gesetzt sein oder nicht und SF ist gleich OF. Hier ist Assemblercode, der auf dieses Bedingungen testet (unter der Annahme, dass EAX vorzeichenbehaftet ist):

```
1       cmp    eax, 5
2       js     signon           ; goto signon wenn SF = 1
3       jo     elseblock        ; goto elseblock wenn SF = 0 und OF = 1
4       jmp    thenblock        ; goto thenblock wenn SF = 0 und OF = 0
5   signon:
6       jo     thenblock        ; goto thenblock wenn SF = 1 und OF = 1
7   elseblock:
8       mov    ebx, 2
9       jmp    next
10  thenblock:
11      mov    ebx, 1
12  next:
```

Der obige Code ist sehr unhandlich. Glücklicherweise besitzt die 80x86 zusätzliche Sprunganweisungen, die diese Art von Tests *viel* einfacher machen. Von jedem gibt es vorzeichenbehaftete und vorzeichenlose Versionen. Tabelle 2.5 zeigt diese Befehle. Die gleich und ungleich Verzweigungen (JE und JNE) sind die selben sowohl für Integer mit Vorzeichen als auch ohne Vorzeichen. (In Wirklichkeit sind JE und JNE wirklich identisch mit jeweils JZ und JNZ.) Jeder der anderen Sprunganweisungen hat zwei Synonyme. Zum Beispiel betrachten wir

mit Vorzeichen		**ohne Vorzeichen**	
JE	Sprung bei `vleft` $=$ `vright`	JE	Sprung bei `vleft` $=$ `vright`
JNE	Sprung bei `vleft` $\neq$ `vright`	JNE	Sprung bei `vleft` $\neq$ `vright`
JL, JNGE	Sprung bei `vleft` $<$ `vright`	JB, JNAE	Sprung bei `vleft` $<$ `vright`
JLE, JNG	Sprung bei `vleft` $\leq$ `vright`	JBE, JNA	Sprung bei `vleft` $\leq$ `vright`
JG, JNLE	Sprung bei `vleft` $>$ `vright`	JA, JNBE	Sprung bei `vleft` $>$ `vright`
JGE, JNL	Sprung bei `vleft` $\geq$ `vright`	JAE, JNB	Sprung bei `vleft` $\geq$ `vright`

Tabelle 2.5: Befehle für Vergleiche mit und ohne Vorzeichen

JL (Jump Less than) und JNGE (Jump Not Greater than or Equal to). Dies sind die gleichen Instruktionen, da:

$$x < y \iff \mathbf{not}(x \geq y)$$

Die vorzeichenlosen Vergleiche verwenden A für *above* und B für *below* anstatt L und G.

Unter Verwendung dieser neuen Sprunganweisungen kann der obige Pseudocode viel leichter in Assembler übersetzt werden.

```
13        cmp     eax, 5
14        jge     thenblock
15        mov     ebx, 2
16        jmp     next
17    thenblock:
18        mov     ebx, 1
19    next:
```

2.2.3 Der `LOOP` Befehl

Die 80x86 stellt mehrere Befehle zur Verfügung, die zur Implementierung von *for*-ähnlichen Schleifen entwickelt wurden. Jeder dieser Befehle benutzt ein Codelabel als seinen einzigen Operanden.

LOOP Dekrementiert ECX und verzweigt zum Label, wenn ECX $\neq 0$

LOOPE, LOOPZ Dekrementiert ECX (das FLAGS Register wird nicht verändert) und verzweigt, wenn ECX $\neq 0$ und ZF $= 1$

LOOPNE, LOOPNZ Dekrementiert ECX (FLAGS unverändert), verzweigt, wenn ECX $\neq 0$ und ZF $= 0$

Die letzten beiden Befehle sind für sequenzielle Suchschleifen nützlich. Der folgende Pseudocode:

```
sum = 0;
for( i = 10; i > 0; i-- )
  sum += i;
```

könnte so in Assemblersprache übersetzt werden:

```
1         mov     eax, 0          ; eax ist sum
2         mov     ecx, 10         ; ecx ist i
3     loop_start:
4         add     eax, ecx
5         loop    loop_start
```

2.3 Die Übersetzung von Standard-Kontrollstrukturen

Dieser Abschnitt betrachtet, wie die Standard-Kontrollstrukturen der Hochsprachen in Assembler implementiert werden können.

2.3.1 If Anweisungen

Der folgende Pseudocode:

```
if ( Bedingung )
  then_block;
else
   else_block ;
```

könnte implementiert werden als:

```
1        ; Code um FLAGS zu setzen
2        jxx    else_block       ; wähle xx für Sprung wenn Bedingung falsch
3        ; Code für then Block
4        jmp    endif
5    else_block:
6        ; Code für else Block
7    endif:
```

Wenn es kein else gibt, dann kann der Sprung zu `else_block` durch einen Sprung zu `endif` ersetzt werden.

```
8        ; Code um FLAGS zu setzen
9        jxx    endif            ; wähle xx für Sprung wenn Bedingung falsch
10       ; Code für then Block
11   endif:
```

2.3.2 While Schleifen

Die *while* Schleife ist eine kopfgesteuerte Schleife:

```
while ( Bedingung ) {
  Rumpf der Schleife;
}
```

Das könnte übersetzt werden zu:

```
1    while:
2        ; Code um FLAGS auf Grundlage der Bedingung zu setzen
3        jxx    endwhile         ; wähle xx für Sprung wenn falsch
4        ; Schleifen-Rumpf
5        jmp    while
6    endwhile:
```

2.3.3 Do while Schleifen

Die *do while* Schleife ist eine fußgesteuerte Schleife:

```
do {
  Rumpf der Schleife;
} while ( Bedingung );
```

Das könnte übersetzt werden zu:

```
1  do:
2        ; Schleifen-Rumpf
3        ; Code um FLAGS auf Grundlage der Bedingung zu setzen
4        jxx    do              ; wähle xx für Sprung wenn wahr
```

2.4 Beispiel: Primzahlsuche

Dieser Abschnitt betrachtet ein Programm, das Primzahlen findet. Um das zu tun, gibt es keine Formel. Erinnern wir uns, dass Primzahlen nur durch 1 und sich selbst ohne Rest teilbar sind. Die grundlegende Methode, die dieses Programm benutzt, ist, die Faktoren aller ungeraden Zahlen[3] unter einer gegebenen Grenze zu finden. Wenn für eine ungerade Zahl kein Faktor gefunden werden kann, dann ist sie prim. Abbildung 2.3 zeigt den grundlegenden Algorithmus, geschrieben in C.

```
1  unsigned guess;   /* laufende Testzahl für Primtest */
2  unsigned factor;  /* möglicher Faktor von guess */
3  unsigned limit;   /* Finde PZ bis zu diesem Wert */
4
5   printf("Find primes up to: ");
6  scanf("%u", &limit);
7   printf("2\n");     /* behandle die ersten beiden */
8   printf("3\n");     /* Primzahlen als Spezialfall */
9  guess = 5;         /* anfängliche Testzahl */
10 while ( guess <= limit ) {
11   /* suche einen Faktor von guess */
12    factor = 3;
13   while ( factor*factor < guess &&
14             guess % factor != 0 )
15      factor += 2;
16    if ( guess % factor != 0 )
17      printf("%d\n", guess);
18   guess += 2;     /* beachte nur ungerade Zahlen */
19 }
```

Abbildung 2.3: Primzahlsuche in C

[3] 2 ist die einzige gerade Primzahl.

Hier ist die Assemblerversion:

prime.asm

```
1  %include "asm_io.inc"
2  segment .data
3  Message db      "Find primes up to: ", 0
4
5  segment .bss
6  Limit   resd    1                   ; finde PZ bis zu dieser Grenze
7  Guess   resd    1                   ; laufende Testzahl für prime
8
9  segment .text
10         global  _asm_main
11 _asm_main:
12         enter   0, 0                ; bereite Routine vor
13         pusha
14
15         mov     eax, Message
16         call    print_string
17         call    read_int            ; scanf("%u", &limit );
18         mov     [Limit], eax
19
20         mov     eax, 2              ; printf("2\n");
21         call    print_int
22         call    print_nl
23         mov     eax, 3              ; printf("3\n");
24         call    print_int
25         call    print_nl
26
27         mov     dword [Guess], 5    ; guess = 5;
28 while_limit:                        ; while ( guess <= limit )
29         mov     eax, [Guess]
30         cmp     eax, [Limit]
31         jnbe    end_while_limit     ; jnbe, da Zahlen ohne VZ sind
32
33         mov     ebx, 3              ; ebx ist factor = 3;
34 while_factor:
35         mov     eax, ebx
36         mul     eax                 ; edx:eax = eax*eax
37         jo      end_while_factor    ; wenn Produkt nicht in eax allein passt
38         cmp     eax, [Guess]
39         jnb     end_while_factor    ; if !(factor*factor < guess)
40         mov     eax, [Guess]
41         mov     edx, 0
42         div     ebx                 ; edx = edx:eax % ebx
43         cmp     edx, 0
44         je      end_while_factor    ; if !(guess % factor != 0)
45
46         add     ebx, 2              ; factor += 2;
47         jmp     while_factor
```

```
48  end_while_factor:
49          je      end_if              ; if !(guess % factor != 0)
50          mov     eax, [Guess]        ; printf("%u\n")
51          call    print_int
52          call    print_nl
53  end_if:
54          add     dword [Guess], 2    ; guess += 2
55          jmp     while_limit
56  end_while_limit:
57
58          popa
59          mov     eax, 0              ; kehre zu C zurück
60          leave
61          ret
```

prime.asm

Kapitel 3

Bitoperationen

3.1 Schiebeoperationen

Assembler erlaubt dem Programmierer die individuellen Bits von Daten zu manipulieren. Eine der einfachen Bitoperationen wird *shift* genannt. Eine Verschiebeoperation verändert die Position der Bits in Daten. Verschiebungen können entweder nach links (d. h. in Richtung der höherwertigen Bits) oder nach rechts (den niederwertigen Bits) sein.

3.1.1 Logische Schiebeoperationen

Eine logische Verschiebung ist der einfachste Typ einer Verschiebung. Sie verschiebt in einer sehr einfachen Weise. Abbildung 3.1 zeigt ein Beispiel einer Verschiebung einer ein-Byte Zahl.

links geschoben	1	1	0	1	0	1	0	0
Original	1	1	1	0	1	0	1	0
rechts geschoben	0	1	1	1	0	1	0	1

Abbildung 3.1: Logische Shifts

Beachte, dass neue, hereinkommende Bits immer Null sind. Die Befehle `SHL` und `SHR` werden benutzt, um logische Verschiebungen nach links bzw. rechts durchzuführen. Diese Befehle erlauben einem, um jede Anzahl von Positionen zu schieben. Die Anzahl der Positionen, um die zu schieben ist, kann entweder eine Konstante sein oder kann im Register `CL` gespeichert werden. Das letzte Bit, das aus dem Datum herausgeschoben wird, wird im Carryflag gespeichert. Hier sind einige Codebeispiele:

```
1  mov    ax, 0C123h
2  shl    ax, 1            ; schiebe 1 bit nach links,  ax = 8246h, CF = 1
3  shr    ax, 1            ; schiebe 1 bit nach rechts, ax = 4123h, CF = 0
4  shr    ax, 1            ; schiebe 1 bit nach rechts, ax = 2091h, CF = 1
5  mov    ax, 0C123h
6  shl    ax, 2            ; schiebe 2 bit nach links,  ax = 048Ch, CF = 1
7  mov    cl, 3
8  shr    ax, cl           ; schiebe 3 bit nach rechts, ax = 0091h, CF = 1
```

3.1.2 Anwendungen der Schiebeoperationen

Schnelle Multiplikation und Division sind die einfachsten Anwendungen der Schiebeoperationen. Erinnern wir uns, dass im Dezimalsystem die Multiplikation und Division mit einer Potenz von zehn einfach ist, es sind nur Ziffern zu verschieben. Das gleiche trifft auf Potenzen von zwei im Binären zu. Um zum Beispiel die binäre Zahl 1011_2 (oder 11 in dezimal) zu verdoppeln, schieben wir einmal nach links um 10110_2 (oder 22) zu erhalten. Der Quotient einer Division durch eine Potenz von zwei ist das Ergebnis einer Schiebung nach rechts. Um einfach durch 2 zu teilen, benutzen wir eine einzelne Rechtsschiebung; um durch 4 (2^2) zu dividieren, schieben wir um 2 Positionen nach rechts; um durch 8 (2^3) zu dividieren, schieben wir 3 Stellen nach rechts, usw. Schiebebefehle sind sehr grundlegend und sind *viel* schneller als die entsprechenden `MUL` und `DIV` Befehle!

Logische Schiebungen können tatsächlich benutzt werden, um vorzeichenlose Werte zu multiplizieren und dividieren. Sie funktionieren im Allgemeinen nicht für Zahlen mit Vorzeichen. Betrachten wir den 2-Byte Wert FFFF (vorzeichenbehaftete -1). Wird er logisch einmal rechts geschoben, ist das Ergebnis 7FFF, das +32 767 ist! Für vorzeichenbehaftete Werte kann ein anderer Typ von Schiebeoperationen verwendet werden.

3.1.3 Arithmetische Schiebeoperationen

Diese Schiebungen wurden entwickelt, damit vorzeichenbehaftete Zahlen schnell mit Potenzen von 2 multipliziert und dividiert werden können. Sie stellen sicher, dass das Vorzeichenbit richtig behandelt wird.

SAL Shift Arithmetic Left - Dieser Befehl ist einfach ein Synonym für `SHL`. Er wird in genau den gleichen Maschinencode übersetzt wie `SHL`. Solange das Vorzeichenbit durch die Schiebung nicht verändert wird, ist das Ergebnis korrekt.

SAR Shift Arithmetic Right - Dies ist ein neuer Befehl, der das Vorzeichenbit (d. h. das MSB) seines Operanden nicht verschiebt. Die anderen Bits werden normal geschoben, außer dass die neuen Bits, die links hereinkommen, Kopien des Vorzeichenbits sind (das heißt, wenn das Vorzeichenbit 1 ist, sind die neuen Bits ebenfalls 1). Folglich werden, wenn ein Byte mit diesem Befehl geschoben wird, nur die unteren 7 Bits geschoben. Wie bei den anderen Schiebungen, wird das letzte heraus geschobene Bit im Carryflag gespeichert.

```
9   mov    ax, 0C123h
10  sal    ax, 1            ; ax = 8246h, CF = 1
11  sal    ax, 1            ; ax = 048Ch, CF = 1
12  sar    ax, 2            ; ax = 0123h, CF = 0
```

3.1.4 Rotierbefehle

Die rotierenden Schiebebefehle arbeiten wie logische Schiebungen, außer dass Bits, die an einem Ende aus dem Datum herausfallen, auf der anderen Seite hinein geschoben werden. Das Datum wird daher wie eine Ringstruktur behandelt. Die zwei einfachsten Rotierbefehle sind `ROL` und `ROR`, die nach links

bzw. nach rechts rotieren. Genauso wie bei den anderen Schiebungen, lassen diese Schiebungen eine Kopie des letzten herumgeschobenen Bits im Carryflag zurück.

```
13      mov    ax, 0C123h
14      rol    ax, 1            ; ax = 8247h, CF = 1
15      rol    ax, 1            ; ax = 048Fh, CF = 1
16      rol    ax, 1            ; ax = 091Eh, CF = 0
17      ror    ax, 2            ; ax = 8247h, CF = 1
18      ror    ax, 1            ; ax = C123h, CF = 1
```

Es gibt zwei zusätzliche Rotierbefehle, die die Bits im Datum und im Carryflag schieben, `RCL` und `RCR` genannt. Wenn zum Beispiel das `AX` Register mit diesen Befehlen rotiert wird, werden die 17 Bits bestehend aus `AX` und dem Carryflag rotiert.

```
19      mov    ax, 0C123h
20      clc                     ; lösche das Carryflag (CF = 0)
21      rcl    ax, 1            ; ax = 8246h, CF = 1
22      rcl    ax, 1            ; ax = 048Dh, CF = 1
23      rcl    ax, 1            ; ax = 091Bh, CF = 0
24      rcr    ax, 2            ; ax = 8246h, CF = 1
25      rcr    ax, 1            ; ax = C123h, CF = 0
```

3.1.5 Eine einfache Anwendung

Hier ist ein Codefragment, das die Anzahl der Bits zählt, die im EAX Register „an“ (d. h. 1) sind.

```
1       mov    bl, 0            ; bl Zähler der Anzahl von ON Bits
2       mov    ecx, 32          ; ecx ist der Schleifenzähler
3   count_loop:
4       shl    eax, 1           ; schiebe Bit ins Carryflag
5       jnc    skip_inc         ; wenn CF == 0, goto skip_inc
6       inc    bl
7   skip_inc:
8       loop   count_loop
```

Der obige Code zerstört den ursprünglichen Wert von `EAX` (`EAX` ist am Ende der Schleife Null). Wenn man den Wert von `EAX` erhalten möchte, kann Zeile 4 durch `rol eax, 1` ersetzt werden.

3.2 Boolesche bitweise Operationen

Es gibt vier allgemeine boolesche Operationen: *AND*, *OR*, *XOR* und *NOT*. Eine *Wahrheitstafel* zeigt das Ergebnis jeder Operation für jeden möglichen Wert seiner Operanden.

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 3.1: Die AND Operation

3.2.1 Die *AND* Operation

Das Ergebnis vom *AND* zweier Bits ist nur 1, wenn beide Bits 1 sind, sonst ist das Ergebnis 0, wie die Wahrheitstafel in Tabelle 3.1 zeigt.

	1	0	1	0	1	0	1	0
AND	1	1	0	0	1	0	0	1
	1	0	0	0	1	0	0	0

Abbildung 3.2: AND auf ein Byte angewandt

Prozessoren unterstützen diese Operationen als Befehle, die unabhängig auf allen Datenbits parallel arbeiten. Wenn zum Beispiel die Inhalte von `AL` und `BL` mit *AND* verknüpft werden, wird die grundlegende *AND* Operation auf jedes der 8 Paare korrespondierender Bits in den beiden Registern angewandt, wie Abbildung 3.2 zeigt. Unten ist ein Codebeispiel:

```
1  mov    ax, 0C123h
2  and    ax, 82F6h        ; ax = 8022h
```

3.2.2 Die *OR* Operation

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 3.2: Die OR Operation

Das inklusive *OR* zweier Bits ist nur 0, wenn beide Bits 0 sind, andernfalls ist das Ergebnis 1, wie die Wahrheitstafel in Tabelle 3.2 zeigt. Unten ist ein Codebeispiel:

```
3  mov    ax, 0C123h
4  or     ax, 0E831h       ; ax = E933h
```

3.2.3 Die *XOR* Operation

Das exklusive *OR* zweier Bits ist genau dann 0, wenn beide Bits gleich sind, sonst ist das Ergebnis 1, wie die Wahrheitstafel in Tabelle 3.3 zeigt. Unten ist ein Codebeispiel:

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 3.3: Die XOR Operation

```
5       mov     ax, 0C123h
6       xor     ax, 0E831h      ; ax = 2912h
```

3.2.4 Die *NOT* Operation

X	NOT X
0	1
1	0

Tabelle 3.4: Die NOT Operation

Die *NOT* Operation ist eine *unäre* Operation (d. h. sie wirkt auf einen Operanden, nicht auf zwei, wie *binäre* Operationen so wie *AND*). Das *NOT* eines Bits ist der invertierte Wert des Bits, wie die Wahrheitstafel in Tabelle 3.4 zeigt. Unten ist ein Codebeispiel:

```
7       mov     ax, 0C123h
8       not     ax              ; ax = 3EDCh
```

Beachte, dass *NOT* das Einerkomplement findet. Im Gegensatz zu den anderen bitweisen Operationen, ändert der `NOT` Befehl kein Bit im `FLAGS` Register.

3.2.5 Der `TEST` Befehl

Der `TEST` Befehl führt eine *AND* Operation durch, aber speichert das Ergebnis nicht. Er setzt nur das `FLAGS` Register auf Grund dessen, was das Ergebnis sein würde (genauso wie der `CMP` Befehl eine Subtraktion durchführt, aber nur `FLAGS` setzt). Wenn zum Beispiel das Ergebnis Null sein würde, würde `ZF` gesetzt werden.

Setze Bit i	*OR* die Zahl mit 2^i (das ist die binäre Zahl, in der nur das Bit i gesetzt ist)
Lösche Bit i	*AND* die Zahl mit der binären Zahl, die nur Bit i gelöscht hat. Dieser Operand wird oft eine *Maske* genannt
Komplementiere Bit i	*XOR* die Zahl mit 2^i

Tabelle 3.5: Verwendung der booleschen Operationen

3.2.6 Anwendungen der Bitoperationen

Bitoperationen sind sehr nützlich, um Datenbits individuell zu manipulieren, ohne die anderen Bits zu verändern. Tabelle 3.5 zeigt drei gängige Verwendungen dieser Operationen. Unten ist etwas Beispielcode, der diese Ideen umsetzt.

```
1  mov    ax, 0C123h
2  or     ax, 8            ; schalte Bit 3 an,    ax = C12Bh
3  and    ax, 0FFDFh       ; schalte Bit 5 ab,    ax = C10Bh
4  xor    ax, 8000h        ; invertiere Bit 31,   ax = 410Bh
5  or     ax, 0F00h        ; schalte Nibble an,   ax = 4F0Bh
6  and    ax, 0FFF0h       ; schalte Nibble ab,   ax = 4F00h
7  xor    ax, 0F00Fh       ; invertiere Nibbles,  ax = BF0Fh
8  xor    ax, 0FFFFh       ; Einerkomplement,     ax = 40F0h
```

Die *AND* Operation kann ebenfalls dazu benutzt werden, den Rest einer Division durch eine Potenz von zwei zu finden. Um den Teilerrest einer Division durch 2^i zu finden, verknüpft man die Zahl durch *AND* mit einer Maske gleich $2^i - 1$. Diese Maske enthält Einsen vom Bit 0 bis zum Bit $i - 1$. Es sind genau diese Bits, die den Rest enthalten. Das Ergebnis des *AND* behält diese Bits und setzt die anderen auf Null. Es folgt ein Codefragment, das den Quotient und den Rest der Division von 100 durch 16 findet.

```
9  mov    eax, 100         ; 100 = 64h
10 mov    ebx, 0000000Fh   ; Maske = 16 - 1 = 15 oder F
11 and    ebx, eax         ; ebx = Rest = 4
12 shr    eax, 4           ; eax = Quotient von eax/2^4 = 6
```

Unter Benutzung des `CL` Registers ist es möglich, beliebige Datenbits zu modifizieren. Es folgt ein Beispiel, das ein beliebiges Bit in `EAX` setzt (anschaltet). Die Nummer des zu setzenden Bits ist in `BH` gespeichert.

```
13 mov    cl, bh           ; bilde zuerst die OR Maske
14 mov    ebx, 1
15 shl    ebx, cl          ; cl mal links schieben
16 or     eax, ebx         ; schalte Bit an
```

Ein Bit abzuschalten ist nur ein bisschen schwieriger.

```
17 mov    cl, bh           ; bilde zuerst die AND Maske
18 mov    ebx, 1
19 shl    ebx, cl          ; cl mal links schieben
20 not    ebx              ; invertiere Bits
21 and    eax, ebx         ; schalte Bit ab
```

Der Code, um ein beliebiges Bit zu komplementieren, sei als Übung für den Leser gelassen.

Es ist nicht ungewöhnlich, den folgenden rätselhaften Befehl in einem 80x86 Programm zu finden.

```
22 xor    eax, eax         ; eax = 0
```

Eine Zahl, mit sich selbst *XOR* verknüpft, ergibt immer Null. Dieser Befehl wird benutzt, da sein Maschinencode kleiner als der entsprechende `MOV` Befehl ist.

3.3 Vermeidung bedingter Sprünge

Moderne Prozessoren benutzen sehr hoch entwickelte Techniken, um Code so schnell wie möglich auszuführen. Eine verbreitete Technik ist als *spekulative Ausführung* bekannt. Diese Technik nutzt die Parallelverarbeitungsmöglichkeiten der CPU, um mehrere Instruktionen auf einmal auszuführen. Bedingte Sprünge stellen für diese Idee ein Problem dar. Im Allgemeinen weiß der Prozessor nicht, ob ein Sprung durchgeführt wird oder nicht. Wird er durchgeführt, wird eine andere Menge an Instruktionen ausgeführt, als wenn er nicht durchgeführt wird. Prozessoren versuchen vorherzusagen, ob der Sprung ausgeführt wird. Wenn die Voraussage falsch ist, hat der Prozessor seine Zeit damit verschwendet, falschen Code auszuführen.

```
1       mov    bl, 0            ; bl Zähler der Anzahl von ON Bits
2       mov    ecx, 32          ; ecx ist der Schleifenzähler
3   count_loop:
4       shl    eax, 1           ; schiebe Bit ins Carryflag
5       adc    bl, 0            ; addiere nur das Carryflag zu bl
6       loop   count_loop
```

Abbildung 3.3: Bits zählen mit `ADC`

Ein Weg, um dieses Problem zu vermeiden, ist, wann immer möglich, die Verwendung bedingter Sprünge zu vermeiden. Der Beispielcode in 3.1.5 (Seite 45) zeigt ein einfaches Beispiel, wo man dies tun könnte. Im vorherigen Beispiel werden die „an“ Bits des EAX Registers gezählt. Es verwendet eine Verzweigung, um den `INC` Befehl zu überspringen. Abbildung 3.3 zeigt, wie die Verzweigung durch Benutzung des `ADC` Befehls entfernt werden kann, um das Carryflag direkt zu addieren.

Die `SETcc` Befehle liefern einen Weg, um Verzweigungen in bestimmten Fällen zu entfernen. Diese Befehle setzen den Wert eines bytegroßen Registers oder Speicheradresse auf Null oder Eins, basierend auf dem Zustand des FLAGS Registers. Die Buchstaben nach `SET` sind die gleichen Buchstaben, die bei den bedingten Sprüngen benutzt werden. Wenn die entsprechende Bedingung von `SETcc` wahr ist, ist das gespeicherte Ergebnis eine Eins, wenn falsch, wird eine Null gespeichert. Zum Beispiel,

```
setz   al               ; AL = 1 wenn Z Flag gesetzt, sonst 0
```

Unter Benutzung dieser Befehle kann man einige clevere Techniken entwickeln, die Werte ohne Verzweigungen berechnen.

Betrachten wird zum Beispiel das Problem, das Maximum zweier Werte zu finden. Der Standardansatz, dieses Problem zu lösen, würde sein, ein `CMP` zu benutzen und einen bedingten Sprung zu verwenden, der darauf reagiert, welcher Wert der größere war. Das folgende Beispielprogramm zeigt, wie das Maximum ohne jegliche Verzweigung gefunden werden kann.

```
1  ; file: max.asm
2  %include "asm_io.inc"
3  segment .data
4
5  message1 db     "Enter a number: ", 0
6  message2 db     "Enter another number: ", 0
7  message3 db     "The larger number is: ", 0
8
9  segment .bss
10
11 input1  resd    1               ; erste eingegebene Zahl
12
13 segment .text
14         global  _asm_main
15 _asm_main:
16         enter   0, 0            ; bereite Routine vor
17         pusha
18
19         mov     eax, message1   ; gebe erste Nachricht aus
20         call    print_string
21         call    read_int        ; lese erste Zahl
22         mov     [input1], eax
23
24         mov     eax, message2   ; gebe zweite Nachricht aus
25         call    print_string
26         call    read_int        ; lese zweite Zahl (in eax)
27
28         xor     ebx, ebx        ; ebx = 0
29         cmp     eax, [input1]   ; vergleiche zweite und erste Zahl
30         setg    bl              ; ebx = (input2 > input1) ?          1 : 0
31         neg     ebx             ; ebx = (input2 > input1) ? 0xFFFFFFFF : 0
32         mov     ecx, ebx        ; ecx = (input2 > input1) ? 0xFFFFFFFF : 0
33         and     ecx, eax        ; ecx = (input2 > input1) ?     input2 : 0
34         not     ebx             ; ebx = (input2 > input1) ?          0 : 0xFFFFFFFF
35         and     ebx, [input1]   ; ebx = (input2 > input1) ?          0 : input1
36         or      ecx, ebx        ; ecx = (input2 > input1) ?     input2 : input1
37
38         mov     eax, message3   ; gebe Ergebnis aus
39         call    print_string
40         mov     eax, ecx
41         call    print_int
42         call    print_nl
43
44         popa
45         mov     eax, 0          ; kehre zu C zurück
46         leave
47         ret
```

Der Trick besteht darin, eine Bitmaske zu schaffen, die benutzt werden kann, um den korrekten Wert für das Maximum auszuwählen. Der `SETG` Befehl in Zeile 30 setzt BL auf 1, wenn die zweite Eingabe das Maximum ist, oder sonst auf 0. Das ist nicht gerade die gewünschte Bitmaske. Um die benötigte Bitmaske zu erzeugen, benutzt Zeile 31 den `NEG` Befehl auf das gesamte EBX Register. (Beachte, dass EBX vorher auf Null gesetzt wurde.) Wenn EBX 0 ist, bewirkt dies nichts; jedoch, wenn EBX 1 ist, ist das Ergebnis die Repräsentation von -1 oder 0xFFFFFFFF im Zweierkomplement. Das ist gerade die benötigte Bitmaske. Der restliche Code verwendet diese Bitmaske, um die richtige Eingabe als Maximum auszuwählen.

Ein alternativer Trick besteht darin, den `DEC` Befehl zu verwenden. Wenn in obigem Beispiel `NEG` durch `DEC` ersetzt wird, wird das Ergebnis wieder entweder 0 oder 0xFFFFFFFF sein. Jedoch sind die Werte gegenüber der Benutzung des `NEG` Befehls vertauscht.

3.4 Bitmanipulationen in C

3.4.1 Die bitweisen Operatoren von C

Anders als einige Hochsprachen stellt C Operatoren für bitweise Operationen bereit. Die *AND* Operation wird durch den binären `&` Operator[1] repräsentiert. Die *OR* Operation wird durch den binären `|` Operator repräsentiert. Die *XOR* Operation wird repräsentiert durch den binären `^` Operator. Und die *NOT* Operation wird durch den unären `~` Operator repräsentiert.

Die Schiebeoperationen werden in C durch die binären `<<` und `>>` Operatoren durchgeführt. Der `<<` Operator führt Linksschiebungen und der `>>` Operator führt Rechtsschiebungen aus. Diese Operatoren haben zwei Operanden. Der linke Operand ist der Wert, der geschoben wird, und der rechte Operand ist die Anzahl von Bits, um die zu schieben ist. Wenn der zu schiebende Wert ein vorzeichenloser Typ ist, wird logisch geschoben. Ist der Wert ein Typ mit Vorzeichen (wie `int`), dann wird arithmetisch geschoben. Unten ist etwas Beispielcode in C, der diese Operatoren verwendet:

```
1  short int s;            /* nimm an, short int ist 16-bit */
2  short unsigned u;
3  s = -1;                 /* s = 0xFFFF (2er Komplement) */
4  u = 100;                /* u = 0x0064 */
5  u = u | 0x0100;         /* u = 0x0164 */
6  s = s & 0xFFF0;         /* s = 0xFFF0 */
7  s = s ^ u;              /* s = 0xFE94 */
8  u = u << 3;             /* u = 0x0B20 (logischer Shift) */
9  s = s >> 2;             /* s = 0xFFA5 (arithmetischer Shift) */
```

3.4.2 Die Verwendung bitweiser Operatoren in C

Die bitweisen Operatoren werden in C zum gleichen Zweck benutzt wie sie in Assembler verwendet werden. Sie erlauben einem, individuelle Datenbits zu manipulieren und können für schnelle Multiplikationen und Divisionen verwendet

[1] Dieser Operator ist verschieden von den binären `&&` und unären `&` Operatoren!

werden. Tatsächlich wird ein schlauer C Compiler automatisch eine Schiebeoperation für eine Multiplikation wie `x *= 2` verwenden.

Macro	Bedeutung
S_IRUSR	user kann lesen
S_IWUSR	user kann schreiben
S_IXUSR	user kann ausführen
S_IRGRP	group kann lesen
S_IWGRP	group kann schreiben
S_IXGRP	group kann ausführen
S_IROTH	others können lesen
S_IWOTH	others können schreiben
S_IXOTH	others können ausführen

Tabelle 3.6: POSIX Makros für Datei-Berechtigungen

Viele APIs[2] von Betriebssystemen (wie *POSIX*[3] und Win32) enthalten Funktionen, die Operanden benutzen, die Daten als Bits kodiert haben. Zum Beispiel unterhalten POSIX-Systeme Dateiberechtigungen für drei verschiedene Typen von Benutzern: *user* (ein besserer Name würde *owner* sein), *group* und *others*. Jedem Benutzertyp kann Erlaubnis gewährt werden eine Datei zu lesen, zu schreiben und/oder auszuführen. Die Berechtigung einer Datei zu ändern, verlangt vom C Programmierer, individuelle Bits zu manipulieren. POSIX definiert als Hilfe verschiedene Makros (siehe Tabelle 3.6). Die Funktion `chmod` kann zum Setzen der Dateiberechtigungen verwendet werden. Diese Funktion braucht zwei Parameter, einen String mit dem Namen der Datei und einen Integer[4] mit den für die gewünschten Berechtigungen entsprechend gesetzten Bits. Zum Beispiel setzt der folgende Code die Berechtigungen, um dem Eigentümer der Datei Lese- und Schreib-, Benutzern in der Gruppe Lese- und den anderen keinen Zugriff zu geben.

```
chmod("foo", S_IRUSR | S_IWUSR | S_IRGRP );
```

Die POSIX `stat` Funktion kann benutzt werden, um die gegenwärtigen Berechtigungsbits für eine Datei herauszufinden. Zusammen mit der `chmod` Funktion benutzt, ist es möglich, einige der Berechtigungen zu modifizieren, ohne andere zu ändern. Hier ein Beispiel, das den Schreibzugriff für andere entfernt und Lesezugriff für den Eigentümer der Datei hinzufügt. Die anderen Berechtigungen werden nicht geändert.

```
1 struct stats file_stats ; /* struct, von stat() verwendet */
2 stat("foo", & file_stats ); /* lese Datei-Info. file_stats.st_mode
3                               enthält die Berechtigungsbits */
4 chmod("foo", (file_stats.st_mode & ~S_IWOTH) | S_IRUSR);
```

[2] Application Programming Interface

[3] Steht für Portable Operating System Interface for Computer Environments. Ein durch die IEEE auf der Basis von UNIX entwickelter Standard.

[4] Tatsächlich ein Parameter vom Typ `mode_t`, der ein typedef zu einem ganzzahligen Typ ist.

3.5 Big and little endian Repräsentationen

Kapitel 1 führte das Konzept der big und little endian Darstellung von Multibyte Daten ein. Jedoch hat der Autor gefunden, dass dieses Thema viele Personen verwirrt. Dieser Abschnitt behandelt das Thema ausführlicher.

Der Leser wird sich erinnern, dass die Bytefolge sich auf die Ordnung bezieht, mit der die individuellen Bytes (*nicht* Bits) eines Multibyte-Datenelements im Speicher abgelegt werden. Big endian ist die einfachste Methode. Sie speichert das höchstwertige Byte zuerst, dann das nächstwertige Byte und so weiter. In anderen Worten, die *großen* Bits werden zuerst gespeichert. Little endian speichert die Bytes in der umgekehrten Reihenfolge (niederwertigste zuerst). Die x86 Prozessorfamilie verwendet die little endian Repräsentation.

Betrachte als Beispiel das Doppelwort, das 12345678_{16} repräsentiert. In big endian Repräsentation würden die Bytes als 12 34 56 78 gespeichert werden. In little endian Repräsentation würden die Bytes als 78 56 34 12 gespeichert werden.

Der Leser fragt sich jetzt wahrscheinlich, warum irgendein vernünftiger Chipdesigner die little endian Repräsentation verwenden sollte? Sind die Ingenieure bei Intel Sadisten, weil sie durch diese verwirrende Repräsentationen einer Vielzahl von Programmierern Leid zufügen? Es scheint, dass die CPU zusätzlichen Aufwand treiben muss, um die Bytes rückwärts im Speicher abzulegen (und die Umkehrung beim Auslesen aus dem Speicher rückgängig zu machen). Die Antwort ist, dass die CPU keinerlei zusätzlichen Aufwand betreibt, wenn sie Speicher im little endian Format liest oder schreibt. Man muss sich klarmachen, dass die CPU aus vielen elektronischen Schaltkreisen aufgebaut ist, die einfach mit Bitwerten arbeiten. Die Bits (und Bytes) sind nicht in irgendeiner notwendigen Reihenfolge in der CPU.

Betrachten wir das 2-Byte `AX` Register. Es kann in die Einzelbyte-Register `AH` und `AL` aufgeteilt werden. Es gibt Schaltkreise in der CPU, die die Werte von `AH` und `AL` halten. Schaltkreise sind in der CPU in keinerlei Reihenfolge. Das bedeutet, dass die Schaltkreise für `AH` nicht vor oder hinter den Schaltkreisen für `AL` sind. Ein `mov` Befehl, der den Wert von `AX` in den Speicher kopiert, kopiert den Wert von `AL`, dann `AH`. Das ist für die CPU kein bisschen schwieriger durchzuführen, als `AH` zuerst zu speichern.

```
1  unsigned short word = 0x1234;
2  unsigned char *p = (unsigned char *) &word;
3
4  if ( p[0] == 0x34 )
5     printf(" Little  Endian Machine\n");
6  else
7     printf("Big Endian Machine\n");
```

Abbildung 3.4: Wie die Bytefolge bestimmt werden kann

Das gleiche Argument lässt sich auf die individuellen Bits in einem Byte anwenden. Sie sind nicht wirklich in irgendeiner Reihenfolge in den Schaltkreisen der CPU (oder Speicher, was dies betrifft). Da jedoch individuelle Bits in CPU oder Speicher nicht adressiert werden können, gibt es keinen Weg, zu wissen

(oder sich darum zu kümmern), in welcher Reihenfolge sie intern in der CPU angeordnet zu sein scheinen.

Der C Code in Abbildung 3.4 zeigt, wie die Bytefolge einer CPU bestimmt werden kann. Der Zeiger p behandelt die Variable word als einen Zeichen-Array mit zwei Elementen. So wird p[0] zum ersten Byte von word im Speicher entwickelt, das von der Bytefolge der CPU abhängt.

3.5.1 Wann man sich um die Bytefolge sorgen muss

Für die typische Programmierung ist die Bytefolge der CPU nicht wesentlich. Am häufigsten wird sie wichtig, wenn binäre Daten zwischen verschiedenen Computersystemen übertragen werden. Das erfolgt gewöhnlich entweder unter Benutzung irgendeines Typs von physikalischem Datenträger (wie einer Disk) oder ein Netzwerk. Da ASCII Daten aus einzelnen Bytes bestehen, ist für sie die Bytefolge kein Thema.

Mit dem Aufkommen von Multibyte-Zeichensätzen wie UNICODE, wird die Byteordnung selbst für Textdaten wichtig. UNICODE unterstützt beide Byteordnungen und besitzt einen Mechanismus, um zu spezifizieren, welche Byteordnung verwendet wird, um die Daten darzustellen.

Alle internen TCP/IP Header speichern Integer im big endian Format (*network byte order* genannt). TCP/IP Bibliotheken stellen C Funktionen zur Verfügung, um mit Angelegenheiten der Bytefolge auf eine portable Weise umgehen zu können. Zum Beispiel konvertiert die Funktion htonl() ein Doppelwort (oder long Integer) vom *host* ins *network* Format. Die Funktion ntohl() führt die gegenteilige Transformation durch.[5] Für ein big endian System geben die beiden Funktionen gerade ihr Argument unverändert zurück. Das ermöglicht einem, Netzwerkprogramme zu schreiben, die auf jedem System, unabhängig von seiner Bytefolge, korrekt übersetzt und laufen werden. Für weitere Informationen über Bytefolge und Netzwerkprogrammierung siehe W. Richard Steven's ausgezeichnetes Buch *UNIX Network Programming*.

```
1  unsigned invert_endian( unsigned x )
2  {
3    unsigned invert;
4    const unsigned char *xp = ( const unsigned char * ) &x;
5    unsigned char *ip = ( unsigned char * ) &invert;
6
7    ip[0] = xp[3];    /* stelle die individuellen Bytes um */
8    ip[1] = xp[2];
9    ip[2] = xp[1];
10   ip[3] = xp[0];
11
12   return invert;    /* gib die umgestellten Bytes zurück */
13 }
```

Abbildung 3.5: invert_endian Funktion

Abbildung 3.5 zeigt eine C Funktion, die die Bytefolge eines Doppelworts umkehrt. Der 486 Prozessor hat einen neuen Maschinenbefehl namens `BSWAP` eingeführt, der die Bytes irgendeines 32 bit Registers umdreht. Zum Beispiel,

[5] In Wirklichkeit stellt die Änderung der Bytefolge eines Integers nur die Bytes um, deshalb sind die Konversionen von big nach little oder little nach big die gleichen Operationen. Folglich machen diese beiden Funktionen das Gleiche.

```
1  bswap   edx             ; vertausche Bytes von edx
```

Die Instruktion kann mit 16 bit Registern nicht verwendet werden. Jedoch kann der `XCHG` Befehl eingesetzt werden, um die Bytes der 16 bit Register, die in 8 bit Register zerlegt werden können, zu tauschen. Zum Beispiel:

```
2  xchg    ah, al          ; vertausche Bytes von ax
```

3.6 Bits zählen

Früher wurde eine einfache Technik angegeben, um die Bits zu zählen, die in einem Doppelwort „an“ sind. Dieser Abschnitt betrachtet andere, weniger direkte Methoden, dies zu tun, als eine Übung, die Bitoperationen, die in diesem Kapitel diskutiert wurden, zu verwenden.

3.6.1 Methode Eins

Die erste Methode ist sehr einfach, aber nicht offensichtlich. Abbildung 3.6 zeigt den Code.

```
1  int count_bits( unsigned int data )
2  {
3    int cnt = 0;
4
5    while( data != 0 ) {
6      data = data & (data - 1);
7      cnt++;
8    }
9    return cnt;
10 }
```

Abbildung 3.6: Bits zählen – Methode Eins

Wie arbeitet diese Methode? Bei jedem Schleifendurchgang wird ein Bit in `data` abgeschaltet. Wenn alle Bits aus sind (d. h. wenn `data` Null ist) wird die Schleife beendet. Die Anzahl der erforderlichen Durchgänge, um `data` Null werden zu lassen, ist gleich der Zahl der Bits im ursprünglichen Wert von `data`.

In Zeile 6 ist die Stelle, an der ein Bit von `data` abgeschaltet wird. Wie funktioniert das? Betrachten wir die allgemeine Form der binären Repräsentation von `data` und die am weitesten rechts stehende 1 in dieser Repräsentation. Nach Definition muss jedes Bit nach dieser 1 Null sein. Nun, was wird die binäre Repräsentation von `data - 1` sein? Die Bits links der am weitesten rechts stehenden 1 werden die gleichen sein wie die für `data`, aber ab dem Punkt der rechtesten 1 werden die Bits das Komplement der originalen Bits in `data` sein. Zum Beispiel:

```
data      =  xxxxx10000
data - 1  =  xxxxx01111
```

wobei die x für beide Zahlen gleich sind. Wenn nun `data` und `data - 1` durch *AND* verknüpft werden, wird das Ergebnis die rechteste 1 in `data` löschen und alle anderen Bits unverändert lassen.

3.6.2 Methode Zwei

Eine Nachschlagetabelle kann ebenfalls benutzt werden um die Bits eines beliebigen Doppelworts zu zählen. Der einfachste Ansatz wäre, die Anzahl der Bits für jedes Doppelwort vorauszuberechnen und diese in einem Array zu speichern. Jedoch gibt es mit diesem Ansatz zwei miteinander verwandte Probleme. Es gibt etwa *4 Milliarden* Doppelwort-Werte! Das bedeutet, dass der Array sehr groß sein wird und dass auch seine Initialisierung sehr zeitaufwendig sein würde. (Tatsächlich, wenn man nicht vorhat, den Array wirklich mehr als 4 Milliarden Mal zu benutzen, wird mehr Zeit benötigt, um den Array zu initialisieren, als benötigt würde, nur die Anzahl Bits unter Benutzung der Methode Eins zu berechnen!)

Eine realistischere Methode würde die Bitzahlen für alle möglichen Bytewerte vorausberechnen und diese in einem Array speichern. Dann kann das Doppelwort in vier Bytewerte aufgespaltet werden. Die Anzahl Bits dieser vier Bytes werden im Array nachgeschlagen und aufsummiert, um die Anzahl Bits im originalen Doppelwort zu finden. Abbildung 3.7 zeigt den Code, um diesen Ansatz zu implementieren.

```
1  static unsigned char byte_bit_count[256];   /* Nachschlagetabelle */
2
3  void   initialize_count_bits  ()
4  {
5    int cnt, i, data;
6
7    for( i = 0; i < 256; i++ ) {
8      cnt = 0;
9      data = i;
10     while( data != 0 ) {     /* Methode Eins */
11       data = data & (data - 1);
12       cnt++;
13     }
14     byte_bit_count[i] = cnt;
15   }
16 }
17
18 int  count_bits( unsigned int data )
19 {
20   const unsigned char *byte = ( unsigned char * ) & data;
21
22   return  byte_bit_count[byte[0]]  +  byte_bit_count[byte[1]]  +
23           byte_bit_count[byte[2]]  +  byte_bit_count[byte[3]];
24 }
```

Abbildung 3.7: Bits zählen – Methode Zwei

Die `initialize_count_bits` Funktion muss vor dem ersten Aufruf der `count_bits` Funktion aufgerufen werden. Diese Funktion initialisiert den globalen `byte_bit_count` Array. Die `count_bits` Funktion sieht die Variable `data` nicht als ein Doppelwort, sondern als einen Array von vier Bytes an. Der `byte` Zeiger wirkt

als Zeiger auf diesen vier-Byte Array. Deshalb ist `byte[0]` eines der Bytes in `data` (entweder das niederwertigste oder das höchstwertige Byte, abhängig davon, ob die Hardware jeweils little oder big endian ist). Natürlich könnte man ein Konstrukt verwenden, wie:

```
(data >> 24) & 0x000000FF
```

um den höchstwertigen Bytewert zu finden und ähnliche für die anderen Bytes; jedoch würden diese Konstrukte langsamer als eine Arrayreferenz sein.

Ein letzter Punkt, es könnte einfach eine `for` Schleife benutzt werden, um die Summe in Zeilen 22 und 23 zu berechnen. Aber eine `for` Schleife würde den Overhead beinhalten, einen Schleifenindex zu initialisieren, den Index nach jeder Iteration zu vergleichen und den Index zu inkrementieren. Die Summe als explizite Summe von vier Werten zu berechnen, wird schneller sein. Tatsächlich würde ein smarter Compiler die Version mit der `for` Schleife zur expliziten Summe umwandeln. Der Prozess, Schleifendurchgänge zu verringern oder zu eliminieren, ist eine Technik der Compileroptimierung und als *Loop unrolling* bekannt.

3.6.3 Methode Drei

Es gibt noch eine weitere clevere Methode, die Bits zu zählen, die in einem Datum gesetzt sind. Diese Methode zählt buchstäblich die Einsen und Nullen des Datums zusammen. Diese Summe muss gleich der Anzahl der Einsen im Datum sein. Betrachten wir als Beispiel, die Einsen in einem Byte, das in einer Variablen namens `data` gespeichert ist, zu zählen. Der erste Schritt besteht darin, die folgende Operation durchzuführen:

```
data = (data & 0x55) + ((data >> 1) & 0x55);
```

Was tut das? Die Hexkonstante `0x55` ist 01010101 in binär. Im ersten Operanden der Addition, wird `data` damit per *AND* verknüpft, Bits an den ungeraden Bitpositionen werden gelöscht. Der zweite Operand `((data >> 1) & 0x55)` bewegt zuerst alle Bits an den geraden Positionen auf eine ungerade Position und benutzt die gleiche Maske, um die gleichen Bits zu löschen. Nun enthält der erste Operand die ungeraden Bits und der zweite Operand die geraden Bits von `data`. Wenn diese beiden Operanden zusammengezählt werden, werden die geraden und ungeraden Bits von `data` zusammengezählt. Wenn zum Beispiel `data` 10110011_2 ist, dann:

$$\begin{array}{r} \texttt{data \& } 01010101_2 \\ +\ \texttt{(data >> 1) \& } 01010101_2 \\ \hline \end{array} \quad \text{oder} \quad \begin{array}{r|cccc} & 00 & 01 & 00 & 01 \\ + & 01 & 01 & 00 & 01 \\ \hline & 01 & 10 & 00 & 10 \end{array}$$

Die Addition rechts zeigt die aktuell zusammengezählten Bits. Die Bits der Bytes sind in vier 2-bit Felder geteilt, um zu zeigen, dass tatsächlich vier unabhängige Additionen durchgeführt werden. Da das größte, das diese Summen sein können, zwei ist, gibt es keine Möglichkeit, dass die Summe ihr Feld überlaufen wird und eine der Summen in den anderen Feldern zerstört.

Natürlich wurde damit noch nicht die gesamte Anzahl Bits berechnet. Jedoch kann die gleiche Technik, die oben angewandt wurde, benutzt werden, um den Gesamtbetrag in einer Reihe ähnlicher Schritte zu berechnen. Der nächste Schritt würde sein:

```
data = (data & 0x33) + ((data >> 2) & 0x33);
```

Führen wir das obige Beispiel weiter (erinnern wir uns, dass `data` jetzt 01100010_2 ist):

`data & 00110011`$_2$			
`+ (data >> 2) & 00110011`$_2$	oder	+	0010 | 0010
			0001 | 0000
			0011 | 0010

Nun gibt es zwei 4-bit Felder, die unabhängig addiert werden.

Der nächste Schritt besteht darin, diese zwei Bitsummen zusammenzuzählen, um das endgültige Ergebnis zu bilden:

```
data = (data & 0x0F) + ((data >> 4) & 0x0F);
```

Unter Benutzung des obigen Beispiels (mit `data` gleich 00110010_2):

`data & 00001111`$_2$			
`+ (data >> 4) & 00001111`$_2$	oder	+	00000010
			00000011
			00000101

Jetzt ist `data` 5, das das korrekte Ergebnis ist. Abbildung 3.8 zeigt eine Implementierung dieser Methode, die die Bits in einem Doppelwort zählt. Sie benutzt eine `for` Schleife, um die Summe zu berechnen. Es würde schneller sein, die Schleife zu entrollen; jedoch macht es die Schleife klarer, wie die Methode sich für verschiedene Datengrößen verallgemeinern lässt.

```
1  int count_bits( unsigned int x )
2  {
3    static unsigned int mask[ ] = { 0x55555555,
4                                    0x33333333,
5                                    0x0F0F0F0F,
6                                    0x00FF00FF,
7                                    0x0000FFFF };
8    int i;
9    int shift;    /* Anzahl Positionen um nach rechts zu schieben */
10
11   for( i=0, shift=1; i < 5; i++, shift *= 2 )
12     x = (x & mask[i]) + ( (x >> shift) & mask[i] );
13   return x;
14 }
```

Abbildung 3.8: Bits zählen – Methode Drei

Kapitel 4

Unterprogramme

Dieses Kapitel betrachtet die Verwendung von Unterprogrammen, um modulare Programme zu erstellen und um mit Hochsprachen (wie C) verwendet zu werden. Funktionen und Prozeduren sind Hochsprachenbeispiele von Unterprogrammen.

Der Code, der ein Unterprogramm aufruft und das Unterprogramm selbst müssen sich einig sein, wie Daten zwischen ihnen ausgetauscht werden. Diese Regeln darüber, wie Daten übergeben werden, werden *Aufrufkonventionen* genannt. Ein großer Teil dieses Kapitels wird sich mit der Standard C Aufrufkonvention befassen, die als Schnittstelle zwischen Unterprogrammen in Assembler und C Programmen dienen kann. Diese (und andere Konventionen) übergeben oft die Adressen von Daten (d. h. Zeiger), um dem Unterprogramm den Zugriff auf die Daten im Speicher zu ermöglichen.

4.1 Indirekte Adressierung

Indirekte Adressierung ermöglicht Registern wie Zeigervariable zu wirken. Um anzuzeigen, dass ein Register indirekt als Zeiger verwendet werden soll, wird es in eckige Klammern (`[]`) eingeschlossen. Zum Beispiel:

```
1  mov    ax, [Data]     ; normale direkte Speicheradressierung eines Worts
2  mov    ebx, Data      ; ebx = &Data
3  mov    ax, [ebx]      ; ax = *ebx
```

Da AX ein Wort enthält, liest Zeile 3 ein Wort, das an der in EBX gespeicherten Adresse beginnt. Wenn AX durch AL ersetzt würde, würde nur ein einzelnes Byte gelesen werden. Es ist wichtig, sich klar zu machen, dass Register keinen Typ wie Variable in C haben. Auf was EBX zeigen soll, wird vollständig dadurch bestimmt, durch welche Befehle es benutzt wird. Weiterhin ist selbst die Tatsache, dass EBX ein Zeiger ist, vollständig dadurch bestimmt, welche Befehle benutzt werden. Wenn EBX falsch benutzt wird, gibt es oft keinen Assemblerfehler; jedoch wird das Programm nicht richtig laufen. Das ist einer der vielen Gründe, dass Assemblerprogrammierung fehleranfälliger als Hochsprachenprogrammierung ist.

Alle 32 bit Allzweck- (EAX, EBX, ECX, EDX) und Index- (ESI, EDI) Register können zur indirekten Adressierung benutzt werden. Im Allgemeinen können die 16 bit und 8 bit Register nicht verwendet werden.

4.2 Einfaches Unterprogramm-Beispiel

Ein Unterprogramm ist eine unabhängige Codeeinheit, die von verschiedenen Teilen eines Programms benutzt werden kann. Mit anderen Worten, ein Unterprogramm ist wie eine Funktion in C. Um ein Unterprogramm aufzurufen, kann ein Sprung benutzt werden, allerdings bereitet die Rückkehr ein Problem. Wenn das Unterprogramm durch verschiedene Teile des Programms benutzt werden soll, muss es zu dem Codeabschnitt zurückkehren, von dem aus es aufgerufen wurde. Folglich kann der Rücksprung vom Unterprogramm nicht hart zu einem Label kodiert sein. Der folgende Code zeigt, wie dies durch Benutzung der indirekten Form des `JMP` Befehls getan werden kann. Diese Form des Befehls verwendet den Wert eines Registers um das Sprungziel zu bestimmen (dabei verhält sich das Register ganz wie ein *Funktionszeiger* in C). Hier ist das erste Programm aus Kapitel 1.4.1 (Seite 17), umgeschrieben, damit es ein Unterprogramm benutzt.

sub1.asm

```
1  ; Datei: sub1.asm
2  ; Unterprogramm Beispielprogramm
3  %include "asm_io.inc"
4
5  segment .data
6  prompt1 db    "Enter a number: ", 0       ; Null Terminator nicht vergessen
7  prompt2 db    "Enter another number: ", 0
8  outmsg1 db    "You entered ", 0
9  outmsg2 db    " and ", 0
10 outmsg3 db    ", the sum of these is ", 0
11
12 segment .bss
13 input1  resd 1
14 input2  resd 1
15
16 segment .text
17         global  _asm_main
18 _asm_main:
19         enter   0, 0              ; bereite Routine vor
20         pusha
21
22         mov     eax, prompt1      ; gib Prompt aus
23         call    print_string
24
25         mov     ebx, input1       ; speichere Adresse von input1 in ebx
26         mov     ecx, ret1         ; speichere Rücksprungadresse in ecx
27         jmp     short get_int     ; lese Integer
28 ret1:
29         mov     eax, prompt2      ; gib Prompt aus
30         call    print_string
31
32         mov     ebx, input2
33         mov     ecx, $ + 7        ; ecx = diese Adresse + 7
```

```
34          jmp     short get_int
35
36          mov     eax, [input1]     ; eax = Dword bei input1
37          add     eax, [input2]     ; eax += Dword bei input2
38          mov     ebx, eax          ; ebx = eax
39
40          mov     eax, outmsg1
41          call    print_string      ; gib erste Nachricht aus
42          mov     eax, [input1]
43          call    print_int         ; gib input1 aus
44          mov     eax, outmsg2
45          call    print_string      ; gib zweite Nachricht aus
46          mov     eax, [input2]
47          call    print_int         ; gib input2 aus
48          mov     eax, outmsg3
49          call    print_string      ; gib dritte Nachricht aus
50          mov     eax, ebx
51          call    print_int         ; gib Summe (ebx) aus
52          call    print_nl          ; gib new-line aus
53
54          popa
55          mov     eax, 0            ; kehre zu C zurück
56          leave
57          ret
58
59  ; Unterprogramm get_int
60  ; Parameter:
61  ;   ebx - Adresse des Dword um den Integer zu speichern
62  ;   ecx - Adresse des Rücksprung-Befehls
63  ; Bemerkung:
64  ;   Wert von eax wird zerstört
65  get_int:
66          call    read_int
67          mov     [ebx], eax         ; speichere Eingabe
68          jmp     ecx                ; springe zum Aufrufer zurück
```

sub1.asm

Das `get_int` Unterprogramm verwendet eine einfache Register basierte Aufrufkonvention. Es erwartet, dass das EBX Register die Adresse des DWORD für die Speicherung der numerischen Eingabe enthält und das ECX Register die Codeadresse der Instruktion, zu der zurück gesprungen werden soll. In Zeilen 25 bis 28 wird das `ret1` Label verwendet, um diese Rücksprungadresse zu berechnen. In Zeilen 32 bis 34 wird der `$` Operator zur Berechnung der Rücksprungadresse verwendet. Der `$` Operator gibt die laufende Adresse der Zeile, in der er erscheint, zurück. Der Ausdruck `$ + 7` berechnet die Adresse des `MOV` Befehls in Zeile 36.

Beide Berechnungen der Rücksprungadresse sind ungeschickt. Die erste Methode erfordert die Definition eines Labels für jeden Unterprogrammaufruf. Die zweite Methode erfordert kein Label, erfordert aber sorgsame Überlegungen. Wenn ein naher statt eines kurzen Sprungs benutzt wird, wird die Zahl, die zu

$ addiert wird, nicht 7 sein! Glücklicherweise gibt es einen viel einfacheren Weg um Unterprogramme aufzurufen. Diese Methode benutzt den *Stack*.

4.3 Der Stack

Viele CPUs verfügen über eine eingebaute Unterstützung für einen Stack. Ein Stack ist eine Last-In First-Out (*LIFO*) Struktur. Der Stack ist ein Speicherbereich, der auf diese Weise organisiert ist. Der `PUSH` Befehl fügt dem Stack Daten hinzu und der `POP` Befehl entnimmt Daten. Das entfernte Datum ist immer das letzte hinzugefügte Datum (deshalb wird er als eine last-in first-out Struktur bezeichnet).

Der Segmentselektor SS spezifiziert das Segment, das den Stack enthält (gewöhnlich ist dies das gleiche Segment, in dem Daten gespeichert werden). Das ESP Register enthält die Adresse des Datums, das aus dem Stack entfernt werden würde. Es wird gesagt, dass sich dieses Datum an der *Spitze* des Stacks (*Top Of the Stack*, *TOS*) befindet. Daten können nur in Doppelwort-Einheiten hinzugefügt werden. Das heißt, man kann kein einzelnes Byte auf den Stack schieben.

Der `PUSH` Befehl fügt dem Stack ein Doppelwort[1] hinzu, indem er 4 von ESP abzieht und dann das Doppelwort nach `[ESP]` speichert. Der `POP` Befehl liest das Doppelwort von `[ESP]` und addiert dann 4 zu ESP. Der folgende Code demonstriert, wie diese Befehle arbeiten und nimmt an, dass ESP anfänglich `1000h` ist.

```
1  push   dword 1        ; 1 gespeichert bei 0FFCh, ESP = 0FFCh
2  push   dword 2        ; 2 gespeichert bei 0FF8h, ESP = 0FF8h
3  push   dword 3        ; 3 gespeichert bei 0FF4h, ESP = 0FF4h
4  pop    eax            ; EAX = 3, ESP = 0FF8h
5  pop    ebx            ; EBX = 2, ESP = 0FFCh
6  pop    ecx            ; ECX = 1, ESP = 1000h
```

Der Stack kann als geeignete Stelle benutzt werden, um Daten temporär zu speichern. Er wird auch benutzt, um Unterprogrammaufrufe zu machen, Parameter zu übergeben und lokale Variable zu speichern.

Die 80x86 stellt auch einen `PUSHA` Befehl bereit, der die Werte der EAX, ECX, EDX, EBX, ESP (originaler Wert), EBP, ESI, und EDI Register (in dieser Reihenfolge) auf den Stack kopiert. Der `POPA` Befehl kann benutzt werden, um sie alle wieder zurückzuspeichern.

4.4 Die `CALL` und `RET` Befehle

Die 80x86 stellt zwei Befehle zur Verfügung, die den Stack benutzen, um den Unterprogrammaufruf schnell und einfach zu machen. Der `CALL` Befehl führt einen unbedingten Sprung zu einem Unterprogramm aus und *speichert* die Adresse des nächsten Befehls auf den Stack. Der `RET` Befehl *holt* sich eine Adresse vom Stack und springt zu dieser Adresse. Wenn man diese Befehle benutzt, ist es

[1] Tatsächlich können auch Wörter auf den Stack geschoben werden, aber im 32-bit protected Mode ist es besser, nur mit Doppelwörtern auf dem Stack zu arbeiten.

sehr wichtig, dass man mit dem Stack richtig umgeht, sodass der richtige Wert durch den `RET` Befehl geholt wird!

Das vorherige Programm kann umgeschrieben werden, um diese neuen Befehle zu benutzen, indem Zeilen 25 bis 34 geändert werden zu:

```
mov    ebx, input1
call   get_int
...
mov    ebx, input2
call   get_int
```

und das Unterprogramm `get_int` geändert wird zu:

```
get_int:
     call   read_int
     mov    [ebx], eax
     ret
```

Mit `CALL` und `RET` gibt es mehrere Vorteile::

- Es ist einfacher!
- Es ermöglicht, Aufrufe von Unterprogrammen einfach zu verschachteln. Beachte, dass `read_int` von `get_int` aufgerufen wird. Dieser Aufruf legt eine weitere Adresse auf den Stack. Am Ende des Codes von `read_int` ist ein RET, das die Rücksprungadresse vom Stack holt und zurück zum Code von `get_int` springt. Dann, wenn das RET von `get_int` ausgeführt wird, holt es die Rücksprungadresse vom Stack, mit der es zu `asm_main` zurückspringt. Dies arbeitet wegen der LIFO Eigenschaft des Stacks korrekt.

Es ist zu beachten, dass es *sehr* wichtig ist, alle Daten vom Stack zu entfernen, die darauf geschoben wurden. Betrachten wir zum Beispiel folgendes:

```
get_int:
     call   read_int
     mov    [ebx], eax
     push   eax
     ret                    ; holt sich Wert von EAX, nicht Rücksprungadresse!!
```

Dieser Code würde nicht richtig zurückkehren!

4.5 Aufrufkonventionen

Wenn ein Unterprogramm aufgerufen wird, müssen der aufrufende Code (*caller*) und das Unterprogramm (*callee*) darin übereinstimmen, wie sie Daten zwischen sich austauschen. Um Daten zu übergeben, verfügen Hochsprachen über Standardverfahren, die als *Aufrufkonventionen* bekannt sind. Um Hochsprachen mit Assembler zu verbinden, muss der Assemblercode die gleichen Konventionen verwenden wie die Hochsprache. Die Aufrufkonventionen können sich von Compiler

zu Compiler unterscheiden oder können davon abhängen, wie der Code kompiliert wird (z. B. ob Optimierungen angeschaltet sind oder nicht). Eine universelle Konvention ist, dass der Code mit einem `CALL` Befehl aufgerufen wird und über ein `RET` zurückkehrt.

Alle PC C Compiler unterstützen eine Aufrufkonvention, die im Rest dieses Kapitels in Etappen beschrieben wird. Diese Konvention ermöglicht es einem, Unterprogramme zu entwickeln, die *wiedereintrittsfähig* (*reentrant*) sind. Ein wiedereintrittsfähiges Unterprogramm kann an jedem Punkt des Programms sicher aufgerufen werden (sogar innerhalb des Unterprogramms selbst).

4.5.1 Parameterübergabe über den Stack

Parameter für ein Unterprogramm können auf dem Stack übergeben werden. Sie werden vor dem `CALL` Befehl auf den Stack gelegt. Genauso wie in C, wenn der Parameter durch das Unterprogramm geändert werden soll, muss die *Adresse* des Datums übergeben werden, nicht der *Wert*. Wenn die Größe des Parameters kleiner als ein Doppelwort ist, muss er zu einem Doppelwort erweitert werden, bevor er übergeben wird.

Die Parameter auf dem Stack werden vom Unterprogramm nicht heruntergenommen, stattdessen wird auf sie im Stack selbst zugegriffen. Warum?

- Da sie vor dem `CALL` Befehl auf den Stack gelegt werden müssen, müsste zuerst die Rücksprungadresse vom Stack genommen (und dann später wieder darauf geschoben) werden.
- Oft werden die Parameter an verschiedenen Stellen des Unterprogramms benutzt werden. Gewöhnlich können sie nicht während des gesamten Unterprogramms in einem Register gehalten werden und müssten im Speicher abgelegt werden. Indem man sie auf dem Stack lässt, hält man eine Kopie der Daten im Speicher, auf die von jedem Punkt des Unterprogramms aus zugegriffen werden kann.

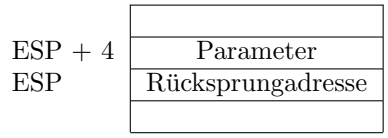


Abbildung 4.1: Stack mit einem Parameter

Betrachten wir ein Unterprogramm, dem ein einzelner Parameter auf dem Stack übergeben wird. Wenn das Unterprogramm aufgerufen wird, sieht der Stack wie in Abbildung 4.1 aus. Auf den Parameter kann unter Verwendung der indirekten Adressierung (`[ESP + 4]`[2]) zugegriffen werden.

Falls der Stack auch innerhalb des Unterprogramms benutzt wird um Daten zu speichern, wird sich die Zahl, die zu ESP addiert wird, ändern. Zum Beispiel zeigt Abbildung 4.2, wie der Stack aussieht, wenn ein DWORD auf den Stack geschoben wurde. Jetzt ist der Parameter bei `ESP + 8`, nicht bei `ESP + 4`. Deshalb kann es sehr fehleranfällig sein, ESP zu benutzen, wenn auf Parameter

[2]Bei der Verwendung der indirekten Adressierung ist es zulässig, eine Konstante zu einem Register zu addieren. Noch kompliziertere Ausdrücke sind auch möglich. Dieses Thema wird im nächsten Kapitel behandelt.

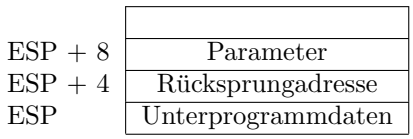


Abbildung 4.2: Stack mit Parameter und lokalen Daten

zugegriffen wird. Um dieses Problem zu lösen, stellt die 80386 ein weiteres Register zur Benutzung zur Verfügung: EBP. Der einzige Zweck dieses Registers ist, Daten auf dem Stack zu referenzieren. Die C Aufrufkonvention fordert, dass ein Unterprogramm zuerst den Wert von EBP auf den Stack sichert und dann EBP gleich ESP setzt. Das ermöglicht, ESP zu ändern, ganz wie Daten auf oder vom Stack genommen werden, ohne EBP zu verändern. Am Ende des Unterprogramms muss der originale Wert von EBP wieder hergestellt werden (deshalb wird er am Anfang des Unterprogramms gesichert). Abbildung 4.3 zeigt die allgemeine Form eines Unterprogramms, das diesen Konventionen folgt.

Wenn indirekte Adressierung benutzt wird, greift der 80x86 Prozessor, in Abhängigkeit davon, welche Register in dem indirekten Adressausdruck benutzt werden, auf verschiedene Segmente zu. ESP (und EBP) benutzen das Stacksegment, während EAX, EBX, ECX und EDX das Datensegment benutzen. Das ist jedoch für die meisten protected Mode Programme unwichtig, da für sie Daten- und Stacksegment gleich sind.

```
1  subprogram_label:
2       push   ebp              ; sichere originalen EBP auf Stack
3       mov    ebp, esp         ; neuer EBP = ESP
4  ; Unterprogramm Code
5       pop    ebp              ; stelle originalen EBP wieder her
6       ret
```

Abbildung 4.3: Allgemeine Form eines Unterprogramms

Zeilen 2 und 3 in Abbildung 4.3 bilden den allgemeinen *Prolog* eines Unterprogramms. Zeilen 5 und 6 bilden den *Epilog*. Abbildung 4.4 zeigt, wie der Stack unmittelbar nach dem Prolog aussieht. Nun kann auf den Parameter von jeder Stelle des Unterprogramm aus mit `[EBP + 8]` zugegriffen werden, ohne sich darum sorgen zu müssen, was sonst noch vom Unterprogramm auf den Stack geschoben wurde.

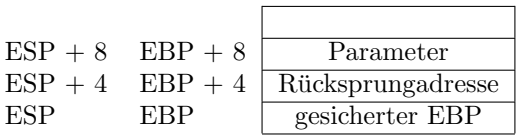


Abbildung 4.4: Stack mit Stackframe

Nachdem das Unterprogramm beendet ist, müssen die auf den Stack geschobenen Parameter entfernt werden. Die Aufrufkonvention von C spezifiziert, dass der aufrufende Code dies tun muss. Andere Konventionen sind verschieden. Zum Beispiel spezifiziert die Aufrufkonvention von Pascal, dass das Unterprogramm die Parameter entfernen muss. (Es gibt eine weitere Form des RET Befehls, die dies zu tun einfach macht.) Einige C Compiler unterstützen auch diese Konvention. Das Schlüsselwort `pascal` wird in Prototyp und Definition der Funktion verwendet, um dem Compiler zu sagen, diese Konvention zu ver-

wenden. Tatsächlich funktioniert die `stdcall` Konvention, die die MS Windows API C Funktionen verwenden, ebenfalls auf diese Art. Was ist der Vorteil dieses Verfahrens? Es ist ein bisschen effizienter als die C Konvention. Warum benutzen dann nicht alle C Funktionen diese Konvention? Im Allgemeinen erlaubt C einer Funktion eine variable Anzahl von Argumenten zu haben (z. B. die `printf` und `scanf` Funktionen). Für diese Art von Funktionen wird sich die Operation, die Parameter vom Stack zu entfernen, von einem Aufruf der Funktion zum nächsten unterscheiden. Die C Konvention ermöglicht den Befehlen, die diese Operation durchführen, leicht von einem Aufruf zum nächsten variiert zu werden. Die Pascal und stdcall Konventionen machen diese Operation sehr schwierig. Folglich erlaubt die Pascal Konvention (wie auch die Pascal Sprache) diese Art von Funktionen nicht. MS Windows kann diese Konvention benutzen, da keine seiner API Funktionen eine variable Anzahl von Argumenten hat.

```
1        push   dword 1        ; übergebe 1 als Parameter
2        call   fun
3        add    esp, 4         ; entferne Parameter vom Stack
```

Abbildung 4.5: Beispiel eines Unterprogrammaufrufs

Abbildung 4.5 zeigt, wie ein Unterprogramm unter Verwendung der C Aufrufkonvention aufgerufen wird. Zeile 3 entfernt durch direkte Manipulation des Stackpointers den Parameter vom Stack. Ein `POP` Befehl könnte ebenso verwendet werden um dies zu tun, würde aber erfordern, dass das nutzlose Ergebnis in ein Register gespeichert wird. Tatsächlich würden in diesem speziellen Fall viele Compiler einen `POP ECX` Befehl benutzen, um den Parameter zu entfernen. Der Compiler würde ein `POP` statt eines `ADD` benutzen, weil das `ADD` mehr Bytes für den Befehl erfordert. Jedoch ändert das `POP` auch den Wert von ECX! Es folgt ein weiteres Beispielprogramm mit zwei Unterprogrammen, die die oben diskutierte C Aufrufkonvention verwenden. Zeile 60 (neben anderen Zeilen) zeigt, dass mehrere Daten- und Textsegmente in einem einzigen Quellprogramm deklariert werden können. Sie werden im Linkprozess zu einem einziges Daten- und Textsegment kombiniert werden. Die Aufteilung von Daten und Code in gesonderte Segmente erlaubt den Daten, die ein Unterprogramm verwendet, in der Nähe des Codes dieses Unterprogramms definiert zu werden.

sub3.asm

```
1   %include "asm_io.inc"
2
3   segment .data
4   sum     dd   0
5
6   segment .bss
7   input   resd 1
8
9   ;
10  ; Algorithmus in Pseudocode
11  ; i = 1;
```

```
12  ; sum = 0;
13  ; while( get_int(i, &input), input != 0 ) {
14  ;   sum += input;
15  ;   i++;
16  ; }
17  ; print_sum(num);
18  ;
19
20  segment .text
21          global  _asm_main
22  _asm_main:
23          enter   0, 0              ; bereite Routine vor
24          pusha
25
26          mov     edx, 1            ; edx ist 'i' im Pseudocode
27  while_loop:
28          push    edx               ; sichere i auf Stack
29          push    dword input       ; lege Adresse von input auf Stack
30          call    get_int
31          add     esp, 8            ; entferne i und &input vom Stack
32
33          mov     eax, [input]
34          cmp     eax, 0
35          je      end_while
36
37          add     [sum], eax        ; sum += input
38
39          inc     edx
40          jmp     short while_loop
41
42  end_while:
43          push    dword [sum]       ; lege Wert von sum auf Stack
44          call    print_sum
45          pop     ecx               ; entferne [sum] vom Stack
46
47          popa
48          leave
49          ret
50
51  ; Unterprogramm get_int
52  ; Parameter (in der Reihenfolge auf dem Stack)
53  ;   Nummer der Eingabe (bei [ebp + 12])
54  ;   Adresse des Worts um die Eingabe zu speichern (bei [ebp + 8])
55  ; Bemerkung:
56  ;   Werte von eax und ebx werden zerstört
57  segment .data
58  prompt  db      ") Enter an integer number (0 to quit): ", 0
59
60  segment .text
61  get_int:
```

```
62          push    ebp
63          mov     ebp, esp
64
65          mov     eax, [ebp+12]
66          call    print_int
67
68          mov     eax, prompt
69          call    print_string
70
71          call    read_int
72          mov     ebx, [ebp+8]
73          mov     [ebx], eax          ; speichere Eingabe
74
75          pop     ebp
76          ret                         ; kehre zum Aufruf zurück
77
78  ; Unterprogramm print_sum
79  ; gibt die Summe aus
80  ; Parameter:
81  ;   auszugebende Summe (bei [ebp+8])
82  ; Bemerkung: zerstört Wert von eax
83  ;
84  segment .data
85  result  db      "The sum is ", 0
86
87  segment .text
88  print_sum:
89          push    ebp
90          mov     ebp, esp
91
92          mov     eax, result
93          call    print_string
94
95          mov     eax, [ebp+8]
96          call    print_int
97          call    print_nl
98
99          pop     ebp
100         ret
```

sub3.asm

4.5.2 Lokale Variable auf dem Stack

Der Stack kann als geeigneter Ort für lokale Variable verwendet werden. Das ist genau da, wo C normale (oder *automatic* in C Lingo) Variable speichert. Die Benutzung des Stacks für Variable ist wichtig, wenn man möchte, dass Unterprogramme wiedereintrittsfähig (reentrant) sind. Ein wiedereintrittsfähiges Unterprogramm wird funktionieren, egal von welcher Stelle aus es aufgerufen wird, einschließlich aus dem Unterprogramm selbst. In anderen Worten können wiedereintrittsfähige Unterprogramm *rekursiv* aufgerufen werden. Die Benutzung

des Stacks für Variable spart auch Speicher. Daten, die nicht auf dem Stack gespeichert werden, belegen Speicher vom Anfang des Programms bis zum Ende des Programms (C nennt diese Typen von Variable *global* oder *static*). Auf dem Stack gespeicherte Daten belegen nur Speicher, wenn das Unterprogramm, in dem sie definiert sind, aktiv ist.

```
1  subprogram_label:
2       push   ebp              ; sichere originalen EBP auf Stack
3       mov    ebp, esp         ; neuer EBP = ESP
4       sub    esp, LOCAL_BYTES ; = # lokal benötigter Bytes
5  ; Unterprogramm Code
6       mov    esp, ebp         ; gebe lokalen Speicher frei
7       pop    ebp              ; stelle originalen EBP wieder her
8       ret
```

Abbildung 4.6: Allgemeine Form eines Unterprogramms mit lokalen Variablen

```
1  void calc_sum( int n, int *sump )
2  {
3    int i, sum = 0;
4
5    for( i=1; i <= n; i++ )
6      sum += i;
7    *sump = sum;
8  }
```

Abbildung 4.7: C Version von sum

Lokale Variable werden direkt nach dem gesicherten Wert von EBP auf dem Stack gespeichert. Ihnen wird Speicher zugeteilt, indem die Anzahl benötigter Bytes im Prolog es Unterprogramms von ESP abgezogen wird. Abbildung 4.6 zeigt das neue Unterprogrammgerüst. Das EBP Register wird benutzt, um auf lokale Variable zuzugreifen. Betrachte die C Funktion in Abbildung 4.7. Abbildung 4.8 zeigt, wie das äquivalente Unterprogramm in Assembler geschrieben werden könnte.

Abbildung 4.9 zeigt, wie der Stack, nach dem Prolog des Programms in Abbildung 4.8, aussieht. Dieser Abschnitt des Stacks, der die Parameter, Rücksprunginformation und Speicher für lokale Variable enthält, wird ein *Stackframe* genannt. Jeder Aufruf einer C Funktion kreïert einen neuen Stackframe auf dem Stack.

Prolog und Epilog eines Unterprogramms können durch Benutzung zweier spezieller Befehle vereinfacht werden, die speziell für diesen Zweck geschaffen wurden. Der `ENTER` Befehl führt den Prolog aus und `LEAVE` den Epilog. Der `ENTER` Befehl hat zwei unmittelbare Operanden. Für die C Aufrufkonvention ist der zweite Operand immer 0. Der erste Operand ist die Anzahl Bytes, die für lokale Variable benötigt wird. Der `LEAVE` Befehl hat keine Operanden. Ab-

Trotz der Tatsache, dass `ENTER` *und* `LEAVE` *den Prolog und Epilog vereinfachen, werden sie nicht sehr oft benutzt. Warum? Weil sie langsamer sind als die gleichwertigen einfacheren Befehle! Das ist ein Beispiel dafür, dass man nicht annehmen kann, dass eine ein-Befehl Sequenz schneller ist als eine mit mehreren Befehlen.*

```
1  cal_sum:
2       push   ebp
3       mov    ebp, esp
4       sub    esp, 4           ; mache Platz für lokale sum
5
6       mov    dword [ebp-4], 0 ; sum = 0
7       mov    ebx, 1           ; ebx (i) = 1
8  for_loop:
9       cmp    ebx, [ebp+12]    ; ist i <= n?
10      jnle   end_for
11
12      add    [ebp-4], ebx     ; sum += i
13      inc    ebx
14      jmp    short for_loop
15
16 end_for:
17      mov    ebx, [ebp+8]     ; ebx = sump
18      mov    eax, [ebp-4]     ; eax = sum
19      mov    [ebx], eax       ; *sump = sum;
20
21      mov    esp, ebp
22      pop    ebp
23      ret
```

Abbildung 4.8: Assembler Version von sum

bildung 4.10 zeigt, wie diese Befehle benutzt werden. Beachte, dass das Programmgerüst (Abbildung 1.7, Seite 23) ebenfalls `ENTER` und `LEAVE` verwendet.

4.6 Programme mit mehreren Modulen

Ein *multi-Modul Programm* ist eins, das aus mehr als einer Objektdatei zusammengesetzt ist. Alle hier vorgestellten Programme sind multi-Modul Programme gewesen. Sie bestanden aus der Objektdatei des C-Treibers und der Assembler Objektdatei (plus den Objektdateien der C Bibliothek). Erinnern wir uns, dass der Linker die Objektdateien zu einem einzigen ausführbaren Programm vereinigt. Der Linker muss die Referenzen zu jedem Label, die in einem Modul (d. h. Objektdatei) bestehen, zu dessen Definition in einem anderen Modul in Verbindung bringen. Damit Modul A ein in Modul B definiertes Label benutzen kann, muss die `extern` Direktive verwendet werden. Nach der Direktive `extern` folgt eine durch Kommata getrennte Liste von Labels. Die Direktive sagt dem Compiler, diese Labels als *extern* zum Modul zu behandeln. Das bewirkt, dass dies Labels sind, die in diesem Modul verwendet werden können, aber in einem anderen definiert sind. Die Datei `asm_io.inc` definiert die `read_int`, usw. Routinen als extern.

ESP + 16	EBP + 12	`n`
ESP + 12	EBP + 8	`sump`
ESP + 8	EBP + 4	Rücksprungadresse
ESP + 4	EBP	gesicherter EBP
ESP	EBP - 4	`sum`

Abbildung 4.9: Stackframe von sum

```
1  subprogram_label:
2       enter  LOCAL_BYTES, 0   ; = # lokal benötigter Bytes
3  ; Unterprogramm Code
4       leave
5       ret
```

Abbildung 4.10: Allgemeine Form eines Unterprogramms mit lokalen Variablen, das `ENTER` und `LEAVE` benutzt

In Assembler kann auf Labels standardmäßig nicht extern zugegriffen werden. Wenn auf ein Label von anderen Modulen aus zugegriffen werden muss, als dem, in dem es definiert ist, muss es in seinem Modul als *global* deklariert werden. Die Direktive `global` bewirkt dies. Zeile 13 des Programmgerüsts in Abbildung 1.7 zeigt, dass das `_asm_main` Label global definiert ist. Ohne diese Deklaration würde es einen Linkerfehler geben. Warum? Weil der C Code nicht in der Lage wäre, sich auf das *interne* `_asm_main` Label zu beziehen.

Es folgt der Code für das vorige Beispiel, zur Benutzung von zwei Modulen umgeschrieben. Die zwei Unterprogramme (`get_int` und `print_sum`) sind in einer von der `_asm_main` Routine getrennten Quelldatei.

main4.asm

```
1   %include "asm_io.inc"
2
3   segment .data
4   sum     dd   0
5
6   segment .bss
7   input   resd 1
8
9   segment .text
10          global  _asm_main
11          extern  get_int, print_sum
12  _asm_main:
13          enter   0, 0              ; bereite Routine vor
14          pusha
15
16          mov     edx, 1            ; edx ist 'i' im Pseudocode
17  while_loop:
18          push    edx               ; sichere i auf Stack
19          push    dword input       ; lege Adresse von input auf Stack
```

```
20          call    get_int
21          add     esp, 8              ; entferne i und &input vom Stack
22
23          mov     eax, [input]
24          cmp     eax, 0
25          je      end_while
26
27          add     [sum], eax          ; sum += input
28
29          inc     edx
30          jmp     short while_loop
31
32  end_while:
33          push    dword [sum]         ; lege Wert von sum auf Stack
34          call    print_sum
35          pop     ecx                 ; entferne [sum] vom Stack
36
37          popa
38          leave
39          ret
```

main4.asm

sub4.asm

```
1   %include "asm_io.inc"
2
3   segment .data
4   prompt  db      ") Enter an integer number (0 to quit): ", 0
5
6   segment .text
7           global  get_int, print_sum
8   get_int:
9           enter   0, 0
10
11          mov     eax, [ebp+12]
12          call    print_int
13
14          mov     eax, prompt
15          call    print_string
16
17          call    read_int
18          mov     ebx, [ebp+8]
19          mov     [ebx], eax          ; speichere Eingabe
20
21          leave
22          ret                         ; springe zum Aufruf zurück
23
24  segment .data
25  result  db      "The sum is ", 0
26
27  segment .text
```

```
28 print_sum:
29         enter   0, 0
30
31         mov     eax, result
32         call    print_string
33
34         mov     eax, [ebp+8]
35         call    print_int
36         call    print_nl
37
38         leave
39         ret
```

sub4.asm

Das vorstehende Beispiel verfügt nur über globale Codelabels; jedoch arbeiten globale Datenlabels auf genau die gleiche Weise.

4.7 Assembler in Verbindung mit C

Heutzutage werden sehr wenige Programme vollständig in Assembler geschrieben. Compiler sind sehr gut darin, Hochsprachencode in effizienten Maschinencode umzusetzen. Da es viel einfacher ist, Code in einer Hochsprache zu schreiben, ist dies populärer. Zusätzlich ist Hochsprachencode *sehr* viel portabler als Assembler!

Wenn Assembler benutzt wird, wird er oft nur für kleine Teile des Codes benutzt. Das kann auf zwei Wege erfolgen: Aufruf von Assemblerroutinen aus C heraus oder inline Assembler. Inline Assembler erlaubt dem Programmierer, Assemblerbefehle direkt in den C Code einzufügen. Das kann sehr bequem sein; jedoch gibt es Nachteile mit inline Assembler. Der Assemblercode muss in dem Format geschrieben werden, den der Compiler verwendet. Zurzeit verwendet kein Compiler das Format von NASM. Verschiedene Compiler erfordern verschiedene Formate. Borland und Microsoft erfordern MASM Format. DJGPP und gcc von Linux erfordern GAS[3] Format. Die Technik, eine Assemblerroutine aufzurufen, ist auf dem PC viel mehr standardisiert.

Assemblerroutinen werden mit C gewöhnlich aus den folgenden Gründen benutzt:

- Direkter Zugriff auf Eigenschaften der Hardware des Computers wird benötigt, bei dem der Zugriff von C aus schwierig oder unmöglich ist.
- Die Routine muss so schnell wie möglich sein und der Programmierer kann den Code besser von Hand optimieren als der Compiler es kann.

Der letzte Grund ist nicht mehr so stichhaltig, wie er einmal war. Die Compilertechnologie hat sich über die Jahre verbessert und Compiler können oft sehr effizienten Code generieren (speziell, wenn Compileroptimierungen angeschaltet sind). Die Nachteile von Assemblerroutinen sind: verminderte Portabilität und Lesbarkeit.

Die meisten der Aufrufkonventionen von C wurden bereits spezifiziert. Jedoch gibt es ein paar zusätzliche Eigenheiten, die beschrieben werden müssen.

[3]GAS ist der Assembler, den alle GNU Compiler verwenden. Er benutzt die AT&T Syntax, die sehr unterschiedlich zu den relativ ähnlichen Syntaxen von MASM, TASM und NASM ist.

```
1  segment .data
2  x            dd     0
3  format       db     "x = %d\n", 0
4
5  segment .text
6  ...
7        push   dword [x]     ; push Wert von x
8        push   dword format  ; push Adresse des Formatstrings
9        call   _printf       ; beachte Unterstrich!
10       add    esp, 8        ; entferne Parameter vom Stack
```

Abbildung 4.11: Aufruf von `printf`

EBP + 12	Wert von `x`
EBP + 8	Adresse des Formatstrings
EBP + 4	Rücksprungadresse
EBP	gesicherter EBP

Abbildung 4.12: Stack innerhalb `printf`

4.7.1 Register sichern

Das Schlüsselwort `register` *kann in einer C Variablendeklaration verwendet werden, um den Compiler darauf hinzuweisen, dass er für diese Variable ein Register anstatt einer Speicherstelle benutzen soll. Sie sind als Registervariable bekannt. Moderne Compiler machen dies automatisch, ohne irgendwelche Vorschläge zu benötigen.*

Zuerst nimmt C an, dass ein Unterprogramm die Werte der folgenden Register erhält: EBX, ESI, EDI, EBP, CS, DS, SS, ES. Das bedeutet nicht, dass das Unterprogramm sie nicht intern ändern kann. Es meint stattdessen, dass, wenn es ihre Werte ändert, es ihre originalen Werte wieder herstellen muss, bevor das Unterprogramm zurückkehrt. Die Werte von EBX, ESI und EDI müssen unverändert bleiben, weil C diese Register für *Registervariable* verwendet. Gewöhnlich wird der Stack verwendet, um die originalen Werte dieser Register zu sichern.

4.7.2 Labels von Funktionen

Die meisten C Compiler stellen ein einzelnes Unterstrichzeichen (_) den Namen von Funktionen und globalen/statischen Variablen voran. Zum Beispiel wird einer Funktion namens `f` das Label `_f` zugeordnet. Folglich, wenn dies eine Assemblerroutine sein soll, *muss* sie `_f` genannt werden, nicht `f`. Der Linux gcc Compiler stellt *keinerlei* Zeichen voran. Unter Linux im ELF Format, würde man einfach das Label `f` für die C Funktion `f` benutzen. Jedoch fügt DJGPPs gcc einen Unterstrich voran. Beachte, dass in der Assembler Programmvorlage (Abbildung 1.7, Seite 23), das Label für die Hauptroutine `_asm_main` ist.

4.7.3 Parameterübergabe

Unter der C Aufrufkonvention werden die Argumente einer Funktion in *umgekehrter* Reihenfolge auf den Stack gelegt als sie im Funktionsaufruf erscheinen.

Betrachten wir das folgende C Statement: `printf("x = %d\n", x);` Abbildung 4.11 zeigt, wie dies kompiliert werden würde (im äquivalenten NASM For-

mat dargestellt). Abbildung 4.12 zeigt, wie der Stack nach dem Prolog innerhalb der `printf` Funktion aussieht. Die `printf` Funktion ist eine der C Bibliotheksfunktionen, die eine beliebige Anzahl von Argumenten haben kann. Die Regeln der Aufrufkonvention von C wurden speziell geschrieben, um diesen Typ von Funktionen zu erlauben. Da die Adresse des Formatstrings zuletzt gespeichert wird, wird sein Platz auf dem Stack *immer* bei `EBP + 8` sein, unabhängig davon, wie viele Parameter an die Funktion übergeben werden. Der Code von `printf` kann dann einen Blick auf den Formatstring werfen, um festzustellen, wie viele Parameter übergeben wurden und sie auf dem Stack erwarten.

Es ist nicht notwendig, Assembler zu verwenden, um eine beliebige Anzahl von Argumenten in C zu benutzen. Die Headerdatei `stdarg.h` *definiert Makros, die benutzt werden können, um sie portabel zu verarbeiten. Siehe jedes gute C Buch für Details.*

Natürlich, wenn der Fehler `printf("x = %d\n")` gemacht wird, wird der `printf` Code immer noch den Doppelwortwert bei `[EBP + 12]` ausdrucken. Jedoch wird das nicht der Wert von `x` sein!

4.7.4 Berechnen der Adressen lokaler Variablen

Die Adresse eines Labels, das im `data` oder `bss` Segment definiert ist, zu finden, ist einfach. Im Grunde macht das der Linker. Jedoch ist die Berechnung der Adresse einer lokalen Variablen (oder Parameter) auf dem Stack nicht so einfach. Das jedoch ist eine sehr alltägliches Aufgabe beim Aufruf von Unterprogrammen. Betrachten wir den Fall, die Adresse einer Variablen (nennen wir sie `x`) an eine Funktion (nennen wir sie `foo`) zu übergeben. Wenn `x` auf dem Stack bei EBP−8 liegt, kann man nicht einfach:

```
mov    eax, ebp-8
```

benutzen. Warum? Der Wert, den `MOV` nach EAX speichert, muss vom Assembler berechnet werden können (das heißt, er muss am Ende eine Konstante sein). Es gibt jedoch einen Befehl, der die benötigte Berechnung (zur Laufzeit) durchführt. Er wird `LEA` (für *Load Effective Address*) genannt. Das Folgende würde die Adresse von `x` berechnen und sie in EAX speichern:

```
lea    eax, [ebp-8]
```

Nun enthält EAX die Adresse von `x` und könnte beim Aufruf der Funktion `foo` auf den Stack geschoben werden. Nicht verwirren lassen, es sieht so aus, als ob der Befehl die Daten von [EBP−8] liest; das ist jedoch *nicht* wahr. Der `LEA` Befehl liest *niemals* aus dem Speicher! Er berechnet nur die Adresse, von der durch andere Befehle gelesen würde und speichert diese Adresse im ersten Registeroperanden. Da er nicht wirklich Speicher ausliest, ist keine Angabe der Speichergröße (z. B. `dword`) nötig oder erlaubt.

4.7.5 Rückgabewerte

Nicht-void C Funktionen geben einen Wert zurück. Die C Aufrufkonventionen spezifizieren wie dies getan wird. Rückgabewerte werden via Register zurückgegeben. Alle ganzzahligen Typen (`char`, `int`, `enum`, usw.) werden im EAX Register zurückgegeben. Wenn sie kürzer als 32 bit sind, werden sie auf 32 bit erweitert, wenn sie in EAX gespeichert werden. (Wie sie erweitert werden, hängt davon ab, ob sie Typen mit oder ohne Vorzeichen sind.) 64 bit Werte werden im EDX:EAX Registerpaar zurückgegeben. Zeigerwerte werden ebenfalls in EAX gespeichert. Fließpunktwerte werden im ST0 Register des mathematischen Coprozessors gespeichert. (Dieses Register wird im Fließpunktkapitel besprochen.)

4.7.6 Andere Aufrufkonventionen

Obige Regeln beschreiben die Standard C Aufrufkonvention, die durch alle 80x86 C Compiler unterstützt wird. Oft unterstützen Compiler auch weitere Aufrufkonventionen. Beim Verwenden von Assembler ist es *sehr* wichtig, zu wissen, welche Aufrufkonvention der Compiler benutzt, wenn er die Funktion aufruft. Für gewöhnlich ist die Voreinstellung, dass die Standardaufrufkonvention benutzt wird; jedoch ist das nicht immer der Fall.[4,5] Compiler, die mehrere Konventionen benutzen, haben oft Kommandozeilenschalter, die verwendet werden können, um die voreingestellte Konvention zu ändern. Sie stellen ebenso Erweiterungen der C Syntax bereit, um individuellen Funktionen explizit Aufrufkonventionen zuzuweisen. Jedoch sind diese Erweiterungen nicht standardisiert und können von einem Compiler zum anderen variieren.

Der GCC Compiler erlaubt verschiedene Aufrufkonventionen. Die Konvention einer Funktion kann expliziert durch die `__attribute__` Erweiterung deklariert werden. Um zum Beispiel eine void Funktion mit Namen `f` zu deklarieren, die die Standard-Aufrufkonvention verwendet und die einen einzelnen `int` Parameter hat, benutzt man die folgende Syntax für ihren Prototyp:

```
void f( int ) __attribute__((cdecl));
```

GCC unterstützt auch die *standard call* Aufrufkonvention. Die obige Funktion könnte unter Benutzung dieser Konvention deklariert werden, indem `cdecl` durch `stdcall` ersetzt wird. Der Unterschied zwischen `stdcall` und `cdecl` ist, dass `stdcall` fordert, dass das Unterprogramm die Parameter vom Stack entfernt (wie es die Aufrufkonvention von Pascal tut). Deshalb kann die `stdcall` Konvention nur mit Funktionen benutzt werden, die eine feste Anzahl von Argumenten benutzt (d. h. nicht mit solchen wie `printf` und `scanf`).

GCC unterstützt auch ein zusätzliches Attribut namens `regparm`, das dem Compiler sagt, Register anstatt den Stack zu benutzen, um bis zu 3 Integerargumente an eine Funktion zu übergeben. Das ist ein allgemeiner Typ von Optimierung, den viele Compiler unterstützen.

Borland und Microsoft benutzen eine gemeinsame Syntax, um Aufrufkonventionen zu deklarieren. Sie fügen die Schlüsselwörter `__cdecl` und `__stdcall` zu C hinzu. Diese Schlüsselwörter wirken als Funktionsmodifizierer und erscheinen unmittelbar vor dem Funktionsnamen in einem Prototypen. Zum Beispiel würde die obige Funktion `f` wie folgt für Borland und Microsoft definiert werden:

```
void __cdecl f( int );
```

Es gibt Vor- und Nachteile für jede der Aufrufkonventionen. Der Hauptvorteil der `cdecl` Konvention ist, dass sie einfach und sehr flexibel ist. Sie kann für jeden Typ von C Funktion und C Compiler verwendet werden. Die Benutzung anderer Konventionen kann die Portabilität des Unterprogramms einschränken. Ihr Hauptnachteil ist, dass sie langsamer als einige der anderen sein kann und mehr Speicher benutzt (da jeder Funktionsaufruf Code erfordert, um die Parameter vom Stack zu entfernen).

Der Vorteil der `stdcall` Konvention ist, dass sie weniger Speicher als `cdecl` benötigt. Hinter dem `CALL` Befehl ist keine Stackbereinigung erforderlich. Ihr

[4] Der Watcom C Compiler ist ein Beispiel für einen, der *nicht* standardmäßig die Standardkonvention benutzt. Siehe die Beispiel-Quellcodedatei für Watcom für Details.

[5] Das gleiche gilt für Delphi, das in der Voreinstellung `register`, nicht `pascal` als Aufrufkonvention verwendet [Anm. d. Ü.]

Hauptnachteil ist, dass sie nicht mit Funktionen verwendet werden kann, die eine variable Anzahl von Argumenten haben.

Der Vorteil, eine Konvention zu verwenden, die Register benutzt um Parameter zu übergeben, ist Geschwindigkeit. Der Hauptnachteil ist, dass die Konvention komplexer ist. Einige Parameter können in Registern sein und andere auf dem Stack.

4.7.7 Beispiele

Es folgt ein Beispiel, das zeigt, wie eine Assemblerroutine mit einem C Programm verknüpft werden kann. (Beachte, dass dieses Programm nicht die Assembler-Dateivorlage (Abbildung 1.7, Seite 23) oder das driver.c Modul verwendet.)

main5.c

```c
#include <stdio.h>
/* Prototyp der Assembler Routine */
void calc_sum( int, int * ) __attribute__((cdecl));

int main( void )
{
  int n, sum;

  printf("Sum integers up to: ");
  scanf("%d", &n);
  calc_sum(n, &sum);
  printf("Sum is %d\n", sum);
  return 0;
}
```

main5.c

sub5.asm

```
; Unterprogramm _calc_sum
; finde die Summe der Integer 1 bis n
; Parameter:
;   n    - Obergrenze der Summation (bei [ebp + 8])
;   sump - Zeiger auf int um sum zu speichern (bei [ebp + 12])
; pseudo C Code:
; void calc_sum( int n, int *sump )
; {
;   int i, sum = 0;
;   for( i=1; i <= n; i++ )
;     sum += i;
;   *sump = sum;
; }

segment .text
        global  _calc_sum
```

```
Sum integers up to: 10
Stack Dump # 1
EBP = BFFFFB70 ESP = BFFFFB68
 +16  BFFFFB80  080499EC
 +12  BFFFFB7C  BFFFFB80
  +8  BFFFFB78  0000000A
  +4  BFFFFB74  08048501
  +0  BFFFFB70  BFFFFB88
  -4  BFFFFB6C  00000000
  -8  BFFFFB68  4010648C
Sum is 55
```

Abbildung 4.13: Beispiellauf des sub5 Programms

```
17  ;
18  ; lokale Variable:
19  ;   sum bei [ebp - 4]
20  _calc_sum:
21          enter   4, 0              ; mache auf Stack Platz für sum
22          push    ebx               ; WICHTIG!
23
24          mov     dword [ebp-4], 0  ; sum = 0
25          dump_stack 1, 2, 4        ; gebe Stack aus von ebp - 8 bis ebp + 16
26          mov     ecx, 1            ; ecx ist i im Pseudocode
27  for_loop:
28          cmp     ecx, [ebp+8]      ; cmp i und n
29          jnle    end_for           ; wenn nicht i <= n, aufhören
30          add     [ebp-4], ecx      ; sum += i
31          inc     ecx
32          jmp     short for_loop
33
34  end_for:
35          mov     ebx, [ebp+12]     ; ebx = sump
36          mov     eax, [ebp-4]      ; eax = sum
37          mov     [ebx], eax
38
39          pop     ebx               ; stelle ebx wieder her
40          leave
41          ret
```

sub5.asm

Warum ist Zeile 22 von `sub5.asm` so wichtig? Weil die C Aufrufkonvention erfordert, dass der Wert von EBX durch den Funktionsaufruf unverändert bleibt. Wird dies nicht beachtet, ist es sehr wahrscheinlich, dass das Programm nicht korrekt arbeiten wird.

Zeile 25 demonstriert, wie das `dump_stack` Makro arbeitet. Erinnern wir uns, dass der erste Parameter nur ein numerisches Label ist und der zweite und dritte Parameter bestimmen, wie viele Doppelwörter unter und über EBP jeweils aus-

gegeben werden sollen. Abbildung 4.13 zeigt einen Beispiellauf des Programms. Aus diesem Speicherauszug kann man entnehmen, dass die Adresse des Dwords für die Speicherung der Summe BFFFFB80 (bei EBP+12) ist; die Zahl, bis zu der zu summieren ist, ist 0000000A (bei EBP+8); die Rücksprungadresse der Routine ist 08048501 (bei EBP+4); der gesicherte Wert von EBP ist BFFFFB88 (bei EBP); der Wert der lokalen Variablen (bei EBP−4) ist 0 und schließlich ist der gesicherte Wert von EBX gleich 4010648C (bei EBP−8).

Die `calc_sum` Funktion könnte umgeschrieben werden, um die Summe als ihren Rückgabewert zurückzugeben, anstatt einen Zeiger als Parameter zu verwenden. Da die Summe ein ganzzahliger Wert ist, sollte die Summe im EAX Register hinterlassen werden. Zeile 11 der Datei `main5.c` müsste geändert werden zu:

```
sum = calc_sum(n);
```

Ebenso müsste der Prototyp von `calc_sum` geändert werden. Unten ist der modifizierte Assemblercode:

sub6.asm

```
1  ; Unterprogramm _calc_sum
2  ; finde die Summe der Integer 1 bis n
3  ; Parameter:
4  ;   n    - Obergrenze der Summation (bei [ebp + 8])
5  ; Rückgabewert:
6  ;   Wert der Summe
7  ; pseudo C Code:
8  ; int calc_sum( int n )
9  ; {
10 ;   int i, sum = 0;
11 ;   for( i=1; i <= n; i++ )
12 ;     sum += i;
13 ;   return sum;
14 ; }
15 segment .text
16         global  _calc_sum
17 ;
18 ; lokale Variable:
19 ;   sum at [ebp-4]
20 _calc_sum:
21         enter   4, 0              ; mache auf Stack Platz für sum
22
23         mov     dword [ebp-4], 0  ; sum = 0
24         mov     ecx, 1            ; ecx ist i im Pseudocode
25 for_loop:
26         cmp     ecx, [ebp+8]      ; cmp i und n
27         jnle    end_for           ; wenn nicht i <= n, aufhören
28
29         add     [ebp-4], ecx      ; sum += i
30         inc     ecx
31         jmp     short for_loop
32
```

```
33  end_for:
34          mov     eax, [ebp-4]      ; eax = sum
35
36          leave
37          ret
```

sub6.asm

4.7.8 Der Aufruf von C Funktionen von Assembler aus

Ein großer Vorteil der Verbindung von C und Assembler ist, dass sie dem Assemblercode gestattet, auf die große Bibliothek von C und auf benutzerdefinierte Funktionen zuzugreifen. Was ist zum Beispiel, wenn man die `scanf` Funktion aufrufen wollte, um einen Integer von der Tastatur einzulesen? Abbildung 4.14 zeigt Code um dies zu tun. Ein sehr wichtiger Punkt zu beachten ist, dass `scanf` dem Aufrufstandard von C bis auf den Buchstaben folgt. Das bedeutet, dass es die Werte von EBX, ESI und EDI erhält; jedoch können die Register EAX, ECX und EDX verändert werden! Tatsächlich wird EAX definitiv geändert, da es den Rückgabewert des `scanf` Aufrufs enthalten wird. Für weitere Beispiele der Verbindung mit C, siehe den Code in `asm_io.asm`, der benutzt wurde, um `asm_io.obj` zu erzeugen.

```
1   segment .data
2   format       db "%d", 0
3
4   segment .text
5   ...
6         lea    eax, [ebp-16]
7         push   eax
8         push   dword format
9         call   _scanf
10        add    esp, 8
11  ...
```

Abbildung 4.14: Aufruf von `scanf` von Assembler

4.8 Reentrante und rekursive Unterprogramme

Ein wiedereintrittsfähiges Unterprogramm muss die folgenden Eigenschaften besitzen:

- Es darf keine Codebefehle verändern. In einer Hochsprache würde dies schwierig sein, aber in Assembler ist es für ein Programm nicht schwer, zu versuchen, seinen eigenen Code zu verändern. Zum Beispiel:

```
1       mov    word [cs:$+7], 5 ; kopiere 5 ins Wort 7 Bytes von hier
2       add    ax, 2            ; vorheriger Befehl ändert 2 in 5!
```

  Dieser Code würde im real Mode arbeiten, aber in protected Mode Betriebssystemen ist das Codesegment als read only markiert. Wenn die erste

Zeile oben ausgeführt wird, wird auf diesen Systemen das Programm abgebrochen. Diese Art von Programmierung ist aus vielen Gründen schlecht. Sie ist verwirrend, schwer zu warten und erlaubt kein Codesharing (siehe unten).

- Es darf keine globalen Daten ändern (wie Daten im `data` und dem `bss` Segment). Alle Variable werden auf dem Stack gespeichert.

Es gibt verschiedene Vorteile, um wiedereintrittsfähigen Code zu schreiben.

- Ein wiedereintrittsfähiges Programm kann rekursiv aufgerufen werden.
- Ein wiedereintrittsfähiges Programm kann von mehreren Prozessen benutzt werden. Auf vielen Multitasking Betriebssystemen ist nur *eine* Kopie des Codes im Speicher, wenn multiple Instanzen eines Programms laufen. Shared Bibliotheken und DLLs (*Dynamic Link Libraries*) nutzen diese Idee genauso.
- Reentrante Unterprogramme arbeiten viel besser in *Multithread*[6]-Programmen. Windows 9x/NT und die meisten Unix-artigen Betriebssysteme (Solaris, Linux, usw.) unterstützen Multithread-Programme.

4.8.1 Rekursive Unterprogramme

Diese Typen von Unterprogrammen rufen sich selbst auf. Die Rekursion kann entweder *direkt* oder *indirekt* sein. Direkte Rekursion tritt auf, wenn ein Unterprogramm, sagen wir `foo`, sich selbst innerhalb `foos` Rumpf aufruft. Indirekte Rekursion tritt auf, wenn ein Unterprogramm nicht direkt durch sich selbst aufgerufen wird, sondern durch ein anderes Unterprogramm, das es aufruft. Zum Beispiel könnte `foo` das Unterprogramm `bar` aufrufen und `bar` könnte `foo` aufrufen.

Rekursive Unterprogramme müssen über eine *Abbruchbedingung* verfügen. Wenn diese Bedingung wahr ist, werden keine weiteren rekursiven Aufrufe gemacht. Wenn eine rekursive Routine keine Abbruchbedingung hat oder die Bedingung niemals wahr wird, wird die Rekursion niemals enden (genauso wie eine unendliche Schleife).

Abbildung 4.15 zeigt eine Funktion, die Fakultäten rekursiv berechnet. Sie könnte von C aufgerufen werden durch:

```
x = fact(3);          /* finde 3! */
```

Abbildung 4.16 zeigt, wie der Stack am tiefsten Punkt des obigen Funktionsaufrufs aussieht.

Abbildungen 4.17 und 4.18 zeigen ein weiteres, komplizierteres rekursives Beispiel jeweils in C und Assembler. Welche Ausgabe gibt `f(3)`? Beachte, dass der `ENTER` Befehl bei jedem rekursiven Aufruf ein neues `i` auf dem Stack anlegt. Deshalb hat jede rekursive Instanz von `f` seine eigene unabhängige Variable `i`. Die Definition von `i` als Doppelwort im `data` Segment würde nicht auf die gleiche Weise funktionieren.

[6]Ein Multithread-Programm besitzt mehrere Ausführungsstränge. Das bedeutet, dass das Programm selbst multitasked ist.

```
1  ; findet n!
2  segment .text
3        global _fact
4  _fact:
5        enter  0, 0
6
7        mov    eax, [ebp+8]    ; eax = n
8        cmp    eax, 1
9        jbe    term_cond       ; beende, wenn n <= 1
10       dec    eax
11       push   eax
12       call   _fact           ; eax = fact(n-1)
13       pop    ecx             ; Antwort in eax
14       mul    dword [ebp+8]   ; edx:eax = eax * [ebp + 8]
15       jmp    short end_fact
16 term_cond:
17       mov    eax, 1
18 end_fact:
19       leave
20       ret
```

Abbildung 4.15: Rekursive Fakultät-Funktion

4.8.2 Wiederholung der Speicherklassen von C

C stellt mehrere Typen von Speicherklassen bereit.

global Diese Variablen werden außerhalb jeder Funktion definiert und werden an festen Speicherplätzen (in den `data` oder `bss` Segmenten) gespeichert und existieren vom Anfang des Programms bis zum Ende. Per Voreinstellung kann auf sie von jeder Funktion im Programm aus zugegriffen werden; wenn sie jedoch `static` deklariert sind, können nur die Funktionen im gleichen Modul auf sie zugreifen (d. h. in der Bezeichnungsweise des Assemblers ist das Label intern, nicht extern).

static Dies sind *lokale* Variable einer Funktion, die `static` deklariert sind. (Unglücklicherweise benutzt C das Schlüsselwort `static` für zwei verschiedene Zwecke!) Diese Variablen liegen ebenfalls an festen Speicherplätzen (in `data` oder `bss`), aber auf sie kann nur direkt durch die Funktionen zugegriffen werden, in denen sie definiert sind.

automatic Dies ist der voreingestellte Typ für eine C Variable, die innerhalb einer Funktion definiert wird. Diese Variablen werden auf dem Stack angelegt, sobald die Funktion, in der sie definiert sind, aufgerufen wird und werden wieder entfernt, wenn die Funktion zurückkehrt. Deshalb haben sie keine festen Speicherplätze.

register Dieses Schlüsselwort bittet den Compiler, für die Daten dieser Variablen ein Register zu verwenden. Das ist nur eine *Anfrage*. Der Compiler muss ihr *nicht* nachkommen. Wenn die Adresse der Variablen irgendwo im Pro-

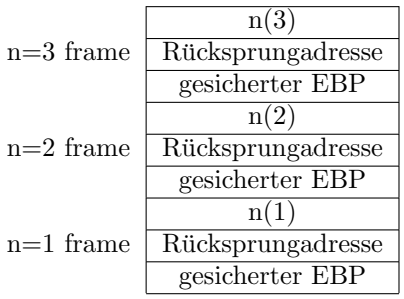


Abbildung 4.16: Stackframes für Fakultäts-Funktion

```
1  void f( int x )
2  {
3    int i;
4    for( i=0; i < x; i++ ) {
5      printf ("%d\n", i);
6      f(i);
7    }
8  }
```

Abbildung 4.17: Ein weiteres Beispiel (C Version)

gramm benutzt wird, wird ihr nicht nachgekommen (da Register keine Adressen haben). Ebenso können nur einfache ganzzahlige Typen Registervariable werden. Strukturierte Typen können keine sein; sie würden nicht in ein Register passen! C Compiler machen oft automatisch normale `automatic` Variable zu Registervariablen ohne jeglichen Hinweis durch den Programmierer.

volatile Dieses Schlüsselwort sagt dem Compiler, dass der Wert der Variablen sich zu jeder Zeit ändern kann. Das bedeutet, dass der Compiler keine Annahmen darüber machen kann, wann die Variable modifiziert wird. Oft könnte ein Compiler den Wert einer Variablen temporär in einem Register speichern und in einem Codeabschnitt das Register anstatt der Variablen benutzen. Diese Art von Optimierung kann er mit `volatile` Variablen nicht durchführen. Ein gebräuchliches Beispiel einer flüchtigen Variablen würde eine sein, die durch zwei Threads eines Multithread-Programms verändert werden kann. Betrachten wir den folgende Code:

```
1  x = 10;
2  y = 20;
3  z = x;
```

Wenn `x` durch einen anderen Thread geändert werden könnte, wäre es möglich, dass der andere Thread `x` zwischen Zeilen 1 und 3 ändert, sodass `z` nicht 10 sein würde. Jedoch, wenn `x` nicht `volatile` definiert wurde, könnte der Compiler annehmen, dass `x` unverändert ist und `z` auf 10 setzen.

Eine weitere Verwendung von `volatile` ist, den Compiler davon abzuhalten, ein Register für eine Variable zu verwenden.

```
1  %define i ebp-4
2  %define x ebp+8            ; nützliche Makros
3  segment .data
4  format  db "%d", 10, 0     ; 10 = '\n'
5  segment .text
6        global _f
7        extern _printf
8  _f:
9        enter  4, 0          ; weise i Platz auf dem Stack zu
10
11       mov    dword [i], 0  ; i = 0
12 lp:
13       mov    eax, [i]      ; ist i < x?
14       cmp    eax, [x]
15       jnl    quit
16
17       push   eax           ; call printf
18       push   format
19       call   _printf
20       add    esp, 8
21
22       push   dword [i]     ; call f
23       call   _f
24       pop    eax
25
26       inc    dword [i]     ; i++
27       jmp    short lp
28 quit:
29       leave
30       ret
```

Abbildung 4.18: Ein weiteres Beispiel (Assembler Version)

Kapitel 5

Arrays

5.1 Einführung

Ein *Array* ist ein zusammenhängender Block einer Liste von Daten im Speicher. Jedes Element der Liste muss den gleichen Typ haben und genau die gleiche Anzahl Bytes für die Speicherung benutzen. Wegen diesen Eigenschaften erlauben Arrays effizienten Zugriff auf die Daten über ihre Position (oder Index) im Array. Die Adresse jeden Elements kann berechnet werden, wenn drei Angaben bekannt sind:

- Die Adresse des ersten Elements des Arrays
- Die Anzahl Bytes in jedem Element
- Der Index des Elements

Es ist bequem, den Index des ersten Arrayelements als Null zu betrachten (genau wie in C). Es ist möglich, andere Werte für den ersten Index zu verwenden, aber es kompliziert die Berechnungen.

5.1.1 Arrays definieren

Arrays im `data` und `bss` Segment definieren

Um einen initialisierten Array im `data` Segment zu definieren, benutzt man die normalen `db`, `dw`, usw. Direktiven. NASM stellt auch eine nützliche Direktive namens `TIMES` zur Verfügung, die verwendet werden kann, um eine Anweisung viele Male zu wiederholen, ohne die Anweisung von Hand duplizieren zu müssen. Abbildung 5.1 zeigt verschiedene Bespiele dazu.

Um einen uninitialisierten Array im `bss` Segment zu definieren, benutzt man die `resb`, `resw`, usw. Direktiven. Erinnern wir uns, dass diese Direktiven einen Operanden haben, der angibt, wie viele Speichereinheiten zu reservieren sind. Abbildung 5.1 zeigt ebenso Beispiele dieses Typs von Definitionen.

Arrays als lokale Variable auf dem Stack definieren

Es gibt keinen direkten Weg, eine lokale Arrayvariable auf dem Stack zu definieren. Wie zuvor berechnet man die gesamte Bytezahl, die für *alle* lokalen

```
1  segment .data
2  ; definiere Array aus 10 Doppelwörtern initialisiert mit 1,2,..,10
3  a1           dd    1, 2, 3, 4, 5, 6, 7, 8, 9, 10
4  ; definiere Array aus 10 Wörtern initialisiert mit 0
5  a2           dw    0, 0, 0, 0, 0, 0, 0, 0, 0, 0
6  ; das Gleiche wie zuvor unter Benutzung von TIMES
7  a3           times 10 dw 0
8  ; definiere Array aus Bytes mit 200 0en und dann 100 1en
9  a4           times 200 db 0
10              times 100 db 1
11
12 segment .bss
13 ; definiere einen Array aus 10 uninitialisierten Doppelwörtern
14 a5           resd  10
15 ; definiere einen Array aus 100 uninitialisierten Wörtern
16 a6           resw  100
```

Abbildung 5.1: Arrays definieren

Variable benötigt werden, einschließlich Arrays und zieht dies von ESP (entweder direkt oder unter Verwendung des `ENTER` Befehls) ab. Wenn eine Funktion zum Beispiel eine Charaktervariable bräuchte, zwei Doppelwortinteger und einen 50-elementigen Wortarray, würde man $1 + 2 \times 4 + 50 \times 2 = 109$ Byte benötigen. Jedoch sollte die von ESP subtrahierte Zahl ein Vielfaches von vier sein (112 in diesem Fall), um ESP auf einer Doppelwortgrenze zu halten. Man könnte die Variablen innerhalb dieser 109 Byte auf verschiedene Weisen anordnen. Abbildung 5.2 zeigt zwei mögliche Arten. Der unbenutzte Teil der ersten Anordnung ist dazu da, die Doppelwörter auf Doppelwortgrenzen zu halten, um die Speicherzugriffe zu beschleunigen.

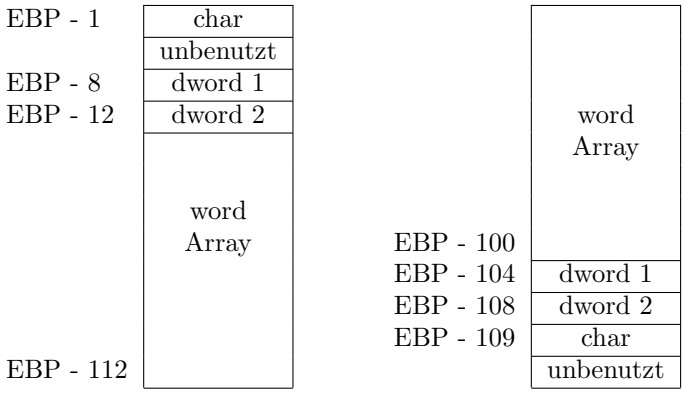


Abbildung 5.2: Anordnungen des Stacks

5.1.2 Auf Elemente des Arrays zugreifen

Es gibt in Assembler keinen `[]` Operator wie in C. Um auf ein Element eines Arrays zuzugreifen, muss seine Adresse berechnet werden. Betrachten wir die folgenden zwei Arraydefinitionen:

```
1 array1       db     5, 4, 3, 2, 1     ; Array von Bytes
2 array2       dw     5, 4, 3, 2, 1     ; Array von Wörtern
```

Hier sind einige Beispiele, die diese Arrays benutzen:

```
3       mov    al, [array1]     ; al = array1[0]
4       mov    al, [array1 + 1] ; al = array1[1]
5       mov    [array1 + 3], al ; array1[3] = al
6       mov    ax, [array2]     ; ax = array2[0]
7       mov    ax, [array2 + 2] ; ax = array2[1] (NICHT array2[2]!)
8       mov    [array2 + 6], ax ; array2[3] = ax
9       mov    ax, [array2 + 1] ; ax = ??
```

In Zeile 7 wird Element 1 des Wortarrays referenziert, nicht Element 2. Warum? Wörter sind zwei-Byte Einheiten, so muss man zwei Bytes weitergehen, um sich zum nächsten Element in einem Wortarray zu bewegen, nicht eins. Zeile 9 liest ein Byte vom ersten Element und eins vom zweiten. In C schaut der Compiler auf den Typ eines Zeigers, um zu bestimmen, wie viele Bytes er in einem Ausdruck, der Zeiger Arithmetik verwendet, voranschreiten muss, sodass es der Programmierer nicht tun muss. Jedoch liegt es in Assembler beim Programmierer, die Größe der Arrayelemente zu berücksichtigen, wenn er sich von Element zu Element bewegt.

```
1       mov    ebx, array1      ; ebx = Adresse von array1
2       mov    dx, 0            ; dx enthält die Summe
3       mov    ah, 0            ; ?
4       mov    ecx, 5
5   lp:
6       mov    al, [ebx]        ; al = *ebx
7       add    dx, ax           ; dx += ax (nicht al!)
8       inc    ebx              ; ebx++
9       loop   lp
```

Abbildung 5.3: Die Elemente eines Arrays zusammenzählen (Version 1)

Abbildung 5.3 zeigt ein Codefragment, das alle Elemente von `array1` aus dem vorigen Beispielcode aufsummiert. In Zeile 7 wird AX zu DX summiert. Warum nicht AL? Erstens müssen die beiden Operanden des `ADD` Befehls von der gleichen Größe sein. Zweitens könnte es leicht passieren, Bytes aufzusummieren und eine Summe zu erhalten, die zu groß war, um in ein Byte zu passen. Indem DX benutzt wird, sind Summen bis hinauf zu 65 535 erlaubt. Es ist jedoch wichtig, sich klar zu machen, dass AH ebenfalls addiert wird. Das ist der Grund, warum AH in Zeile 3 auf Null[1] gesetzt wurde.

```
1        mov    ebx, array1      ; ebx = Adresse von array1
2        mov    dx, 0            ; dx enthält die Summe
3        mov    ecx, 5
4    lp:
5        add    dl, [ebx]        ; dl += *ebx
6        jnc    next             ; if no carry goto next
7        inc    dh               ; inc dh
8    next:
9        inc    ebx              ; ebx++
10       loop   lp
```

Abbildung 5.4: Die Elemente eines Arrays zusammenzählen (Version 2)

```
1        mov    ebx, array1      ; ebx = Adresse von array1
2        mov    dx, 0            ; dx enthält die Summe
3        mov    ecx, 5
4    lp:
5        add    dl, [ebx]        ; dl += *ebx
6        adc    dh, 0            ; dh += carry flag + 0
7        inc    ebx              ; ebx++
8        loop   lp
```

Abbildung 5.5: Die Elemente eines Arrays zusammenzählen (Version 3)

Abbildungen 5.4 und 5.5 zeigen zwei alternative Wege, um die Summe zu berechnen. Die Zeilen in Schrägschrift ersetzen Zeilen 6 und 7 von Abbildung 5.3.

5.1.3 Fortgeschrittenere indirekte Adressierung

Es dürfte nicht überraschen, dass indirekte Adressierung oft mit Arrays verwendet wird. Die allgemeinste Form einer indirekten Speicherreferenz ist:

```
[ base reg + factor * index reg + constant]
```

wobei:

base reg eines der Register EAX, EBX, ECX, EDX, EBP, ESP, ESI oder EDI ist.

factor ist entweder 1, 2, 4 oder 8. (Wenn 1, wird `factor` weggelassen.)

index reg ist eines der Register EAX, EBX, ECX, EDX, EBP, ESI, EDI. (Beachte, dass ESP nicht in der Liste ist.)

constant ist eine 8- oder 32-bit Konstante. Die Konstante kann ein Label (oder ein Labelausdruck) sein.

[1] Indem AH auf Null gesetzt wird, wird implizit angenommen, dass AL eine vorzeichenlose Zahl ist. Wenn sie vorzeichenbehaftet wäre, würde die passende Aktion sein, stattdessen einen `CBW` Befehl zwischen Zeilen 6 und 7 einzufügen.

5.1.4 Beispiel

Hier ist ein Beispiel, das einen Array benutzt und ihn an eine Funktion übergibt. Es benutzt als Treiber das `array1c.c` Programm (unten aufgeführt), nicht das `driver.c` Programm.

array1.asm

```
1  %define ARRAY_SIZE 100
2  %define NEW_LINE 10
3
4  segment .data
5  FirstMsg        db   "First 10 elements of array", 0
6  Prompt          db   "Enter index of element to display: ", 0
7  SecondMsg       db   "Element %d is %d", NEW_LINE, 0
8  ThirdMsg        db   "Elements 20 through 29 of array", 0
9  InputFormat     db   "%d", 0
10
11 segment .bss
12 array   resd    ARRAY_SIZE
13
14 segment .text
15         extern  _puts, _printf, _scanf, _dump_line
16         global  _asm_main
17 _asm_main:
18         enter   4, 0                  ; lokale Dword Variable bei EBP - 4
19         push    ebx
20         push    esi
21
22 ; initialisiere Array mit 100, 99, 98, 97, ...
23
24         mov     ecx, ARRAY_SIZE
25         mov     ebx, array
26 init_loop:
27         mov     [ebx], ecx
28         add     ebx, 4
29         loop    init_loop
30
31         push    dword FirstMsg        ; gebe FirstMsg aus
32         call    _puts
33         pop     ecx
34
35         push    dword 10
36         push    dword array
37         call    _print_array          ; gebe erste 10 Elemente von array aus
38         add     esp, 8
39
40 ; frage Benutzer nach Index des Elements
41 Prompt_loop:
42         push    dword Prompt
43         call    _printf
44         pop     ecx
```

```
45
46          lea     eax, [ebp-4]            ; eax = Adresse des lokalen Dwords
47          push    eax
48          push    dword InputFormat
49          call    _scanf
50          add     esp, 8
51          cmp     eax, 1                  ; eax = Rückgabewert von scanf
52          je      InputOK
53
54          call    _dump_line              ; bei ungültiger Eingabe verwerfe Rest
55          jmp     Prompt_loop             ; der Zeile und beginne nochmals
56
57  InputOK:
58          mov     esi, [ebp-4]
59          push    dword [array + 4*esi]
60          push    esi
61          push    dword SecondMsg         ; gebe Wert des Elements aus
62          call    _printf
63          add     esp, 12
64
65          push    dword ThirdMsg          ; gebe Elemente 20-29 aus
66          call    _puts
67          pop     ecx
68
69          push    dword 10
70          push    dword array + 20*4      ; Adresse von array[20]
71          call    _print_array
72          add     esp, 8
73
74          pop     esi
75          pop     ebx
76          mov     eax, 0                  ; kehre zu C zurück
77          leave
78          ret
79
80  ;
81  ; Routine _print_array
82  ; Von C aufrufbare Routine die die Elemente eines Doppelwort-Arrays
83  ; als Integer mit Vorzeichen ausgibt.
84  ; C Prototyp:
85  ; void print_array( const int *a, int n );
86  ; Parameter:
87  ;   a - Zeiger zum auszugebenden Array (bei ebp + 8 auf Stack)
88  ;   n - Anzahl auszugebender Integer (bei ebp + 12 auf Stack)
89
90  segment .data
91  OutputFormat    db   "%-5d %5d", NEW_LINE, 0
92
93  segment .text
94          global  _print_array
```

```
95  _print_array:
96          enter   0, 0
97          push    esi
98          push    ebx
99
100         xor     esi, esi                ; esi = 0
101         mov     ecx, [ebp + 12]         ; ecx = n
102         mov     ebx, [ebp + 8]          ; ebx = Adresse des Arrays
103 print_loop:
104         push    ecx                     ; printf könnte ecx ändern!
105
106         push    dword [ebx + 4*esi]     ; push array[esi]
107         push    esi
108         push    dword OutputFormat
109         call    _printf
110         add     esp, 12                 ; entferne Parameter (lasse ecx!)
111
112         inc     esi
113         pop     ecx
114         loop    print_loop
115
116         pop     ebx
117         pop     esi
118         leave
119         ret
```

array1.asm

array1c.c

```
1   #include <stdio.h>
2
3   int asm_main( void );
4   void dump_line( void );
5
6   int main()
7   {
8     int ret_status ;
9     ret_status = asm_main();
10    return ret_status ;
11  }
12
13  /*
14   * Funktion dump_line
15   * verwirft alle im Eingabepuffer übrig gebliebenen Zeichen
16   */
17  void dump_line()
18  {
19    int ch;
20
21    while( (ch = getchar()) != EOF && ch != '\n')
```

```
22      /* leerer Rumpf */;
23  }
```

array1c.c

Nochmals der `LEA` Befehl

Der `LEA` Befehl kann noch für weitere Aufgaben verwendet werden, als nur Adressen zu berechnen. Eine ziemlich einfache ist für schnelle Berechnungen. Betrachten wir das Folgende:

```
lea    ebx, [4*eax + eax]
```

Dies speichert effektiv den Wert von $5 \times$ `EAX` in `EBX`. Die Verwendung von `LEA` für diesen Zweck ist sowohl einfacher als auch schneller als die Verwendung von `MUL`. Jedoch muss man sich klarmachen, dass der Ausdruck innerhalb der eckigen Klammern eine gültige indirekte Adresse sein *muss*. Deshalb kann dieser Befehl zum Beispiel nicht verwendet werden, um schnell mit 6 zu multiplizieren.

5.1.5 Mehrdimensionale Arrays

Mehrdimensionale Arrays unterscheiden sich nicht wirklich sehr stark von den bereits betrachteten einfachen eindimensionalen Arrays. Tatsächlich werden sie im Speicher als genau das repräsentiert, als ein einfacher eindimensionaler Array.

Zweidimensionale Arrays

Nicht überraschend ist der einfachste mehrdimensionale Array ein zweidimensionaler. Ein zweidimensionaler Array wird oft als Gitter von Elementen dargestellt. Jedes Element wird durch ein Paar von Indizes identifiziert. Per Übereinkunft wird der erste Index mit der Reihe des Elements identifiziert und der zweite Index mit der Spalte.

Betrachten wir einen Array mit drei Reihen und zwei Spalten, der definiert ist als:

```
int a [3][2];
```

Der C Compiler würde Platz für einen 6 ($= 2 \times 3$) elementigen Integerarray reservieren und die Elemente wie folgt anlegen:

Index	0	1	2	3	4	5
Element	a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

Was die Tabelle zu zeigen versucht, ist, dass das Element, auf das mit `a[0][0]` zugegriffen wird, am Anfang des 6-elementigen eindimensionalen Arrays gespeichert wird. Element `a[0][1]` wird an der nächsten Position (Index 1) gespeichert und so weiter. Jede Reihe des zweidimensionalen Arrays wird fortlaufend im Speicher abgelegt. Das letzte Element einer Reihe wird vom ersten Element der nächsten Reihe gefolgt. Das ist als eine *reihenweise* Repräsentation des Arrays bekannt und ist, wie ein C/C++ Compiler einen Array repräsentieren würde.

Wie bestimmt der Compiler, wo `a[i][j]` in einer reihenweisen Repräsentation erscheint? Eine einfache Formel berechnet den Index aus `i` und `j`. Die Formel ist in diesem Fall $2i + j$. Es ist nicht zu schwer zu sehen, wovon sich diese

Formel ableitet. Jede Zeile ist zwei Elemente lang; so liegt das erste Element von Reihe i an der Stelle $2i$. Dann wird die Position von Spalte j gefunden, indem j zu $2i$ addiert wird. Diese Analyse zeigt auch, wie die Formel für einen Array mit `N` Spalten verallgemeinert wird: $N \times i + j$. Beachte, dass die Formel *nicht* von der Anzahl der Reihen abhängt.

```
1       mov    eax, [ebp-44]       ; ebp - 44 ist i's Platz
2       sal    eax, 1              ; multipliziere i mit 2
3       add    eax, [ebp-48]       ; addiere j
4       mov    eax, [ebp+4*eax-40] ; ebp - 40 ist die Adresse von a[0][0]
5       mov    [ebp-52], eax       ; speichere Ergebnis in x (bei ebp - 52)
```

Abbildung 5.6: Assemblercode für `x = a[i][j]`

Als ein Beispiel werden wir uns ansehen, wie *gcc* den folgenden Code kompiliert (unter Verwendung des oben definierten Arrays `a`):

```
x = a[i][j];
```

Abbildung 5.6 zeigt den Assemblercode, in den dies übersetzt wurde. Somit konvertiert der Compiler den Code im Wesentlichen zu:

```
x = *(&a[0][0] + 2*i + j);
```

und in der Tat könnte der Programmierer ihn in dieser Weise mit demselben Ergebnis schreiben.

Es ist nichts Magisches an der Wahl der reihenweisen Repräsentation des Arrays. Eine spaltenweise Repräsentation würde genauso gut arbeiten:

Index	0	1	2	3	4	5
Element	a[0][0]	a[1][0]	a[2][0]	a[0][1]	a[1][1]	a[2][1]

In der *spaltenweisen* Repräsentation wird jede Spalte fortlaufend gespeichert. Element `[i][j]` wird an Position $i + 3j$ gespeichert. Andere Sprachen (FORTRAN zum Beispiel[2]) benutzen die spaltenweise Repräsentation. Das ist wichtig, wenn man Code mit mehreren Sprachen verbindet.

Dimensionen über zwei

Bei Dimensionen über zwei wird die gleiche grundlegende Idee angewandt. Betrachten wir einen dreidimensionalen Array:

```
int b [4][3][2];
```

Dieser Array würde gespeichert, wie wenn er vier zweidimensionale Arrays, jeder mit Größe `[3][2]` fortlaufend im Speicher wäre. Die unten stehende Tabelle zeigt, wie er beginnt:

[2] mit 1- statt 0-basierten Indizes [Anm. d. Ü.]

Index	0	1	2	3	4	5
Element	b[0][0][0]	b[0][0][1]	b[0][1][0]	b[0][1][1]	b[0][2][0]	b[0][2][1]
Index	6	7	8	9	10	11
Element	b[1][0][0]	b[1][0][1]	b[1][1][0]	b[1][1][1]	b[1][2][0]	b[1][2][1]

Die Formel, um die Position von `b[i][j][k]` zu berechnen, ist $6i + 2j + k$. Die 6 ist gegeben durch die Größe des `[3][2]` Arrays. Im Allgemeinen wird die Position des Elements `a[i][j][k]` in einem als `a[L][M][N]` dimensionierten Array $M \times N \times i + N \times j + k$ sein. Beachte wieder, dass die erste Dimension (`L`) nicht in der Formel erscheint.

Für höhere Dimensionen wird derselbe Prozess generalisiert. Für einen n dimensionalen Array mit Dimensionen D_1 bis D_n ist die Position des durch die Indizes i_1 bis i_n bezeichneten Elements durch die Formel:

$$D_2 \times D_3 \cdots \times D_n \times i_1 + D_3 \times D_4 \cdots \times D_n \times i_2 + \cdots + D_n \times i_{n-1} + i_n$$

gegeben oder für die Mathefreaks kann es prägnanter geschrieben werden als:

$$\sum_{j=1}^{n} \left(\prod_{k=j+1}^{n} D_k \right) i_j$$

Hier ist die Stelle, an der man erkennen kann, dass der Autor Physik als Hauptfach hatte. (Oder hat ihn die Erwähnung von FORTRAN verraten?)

Die erste Dimension, D_1, tritt in der Formel nicht auf.

Für die spaltenweise Repräsentation, wäre die allgemeine Formel:

$$i_1 + D_1 \times i_2 + \cdots + D_1 \times D_2 \times \cdots \times D_{n-2} \times i_{n-1} + D_1 \times D_2 \times \cdots \times D_{n-1} \times i_n$$

oder in der Notation für Mathefreaks:

$$\sum_{j=1}^{n} \left(\prod_{k=1}^{j-1} D_k \right) i_j$$

In diesem Fall ist es die letzte Dimension, D_n, die in der Formel nicht auftritt.

Die Übergabe mehrdimensionaler Arrays als Parameter in C

Die reihenweise Repräsentation mehrdimensionaler Arrays hat einen direkten Einfluss auf die C Programmierung. Für eindimensionale Arrays wird die Größe des Arrays nicht benötigt, um zu berechnen, wo irgendein spezifisches Element im Speicher liegt. Das trifft auf mehrdimensionale Arrays nicht zu. Um auf die Elemente dieser Arrays zuzugreifen, muss der Compiler alle außer der ersten Dimension kennen. Dies wird offenbar, wenn man den Prototypen einer Funktion betrachtet, die einen mehrdimensionalen Array als Parameter hat. Das Folgende wird nicht kompiliert:

```
void f( int a[ ][ ] );   /* keine Dimensionsinformation */
```

Jedoch wird das Folgende kompiliert:

```
void f( int a[ ][2] );
```

LODSB AL = [DS:ESI] ESI = ESI ± 1	STOSB [ES:EDI] = AL EDI = EDI ± 1
LODSW AX = [DS:ESI] ESI = ESI ± 2	STOSW [ES:EDI] = AX EDI = EDI ± 2
LODSD EAX = [DS:ESI] ESI = ESI ± 4	STOSD [ES:EDI] = EAX EDI = EDI ± 4

Abbildung 5.7: Lesende und schreibende Stringbefehle

Jeder zweidimensionale Array mit zwei Spalten kann an diese Funktion übergeben werden. Die erste Dimension wird nicht benötigt.[3]

Nicht verwirren lassen durch eine Funktion mit diesem Prototypen:

```
void f( int *a[ ] );
```

Dies definiert einen eindimensionalen Array von Integerzeigern (der nebenbei dazu verwendet werden kann, um einen Array von Arrays zu schaffen, der sich ganz so wie ein zweidimensionaler Array verhält).

Für höherdimensionale Arrays müssen bei Parametern alle außer der ersten Dimension angegeben werden. Zum Beispiel könnte ein vierdimensionaler Array so übergeben werden:

```
void f( int a[ ][4][3][2]  );
```

5.2 Array/String Befehle

Die 80x86 Familie von Prozessoren stellt verschiedene Befehle, die für die Arbeit mit Arrays geschaffen wurden, zur Verfügung. Diese Befehle werden *Stringbefehle* genannt. Sie benutzen die Indexregister (ESI und EDI) um eine Operation durchzuführen und erhöhen oder vermindern dann automatisch eines oder beide der Indexregister. Das *Richtungsflag* (*direction flag*, DF) im FLAGS Register bestimmt, ob die Indexregister erhöht oder vermindert werden. Es gibt zwei Befehle, die das Richtungsflag ändern:

CLD löscht das Richtungsflag. In diesem Zustand werden die Indexregister erhöht.

STD setzt das Richtungsflag. In diesem Zustand werden die Indexregister vermindert.

Ein *sehr* verbreitertes Versehen in der 80x86 Programmierung ist, zu vergessen, das Richtungsflag explizit in den richtigen Zustand zu setzen. Das führt oft zu Code, der die meiste Zeit funktioniert (wenn sich das Richtungsflag zufällig im gewünschten Zustand befindet), aber er funktioniert nicht *immer*.

5.2.1 Speicherbereiche lesen und schreiben

Die einfachsten Stringbefehle lesen entweder aus oder schreiben in den Speicher oder beides. Sie können auf einmal ein Byte, Wort oder Doppelwort lesen oder schreiben. Abbildung 5.7 zeigt diese Befehle mit einer kurzen Beschreibung in

[3] Eine Größe kann hier angegeben werden, wird aber vom Compiler ignoriert.

```
1  segment .data
2  array1  dd     1, 2, 3, 4, 5, 6, 7, 8, 9, 10
3
4  segment .bss
5  array2  resd   10
6
7  segment .text
8          cld                     ; dies nicht vergessen!
9          mov    esi, array1
10         mov    edi, array2
11         mov    ecx, 10
12 lp:
13         lodsd
14         stosd
15         loop   lp
```

Abbildung 5.8: Load und store Beispiel

MOVSB	byte [ES:EDI] = byte [DS:ESI] ESI = ESI ± 1 EDI = EDI ± 1
MOVSW	word [ES:EDI] = word [DS:ESI] ESI = ESI ± 2 EDI = EDI ± 2
MOVSD	dword [ES:EDI] = dword [DS:ESI] ESI = ESI ± 4 EDI = EDI ± 4

Abbildung 5.9: Die Memory move String Befehle

Pseudocode dessen was sie tun. Es gibt hier verschiedene Punkte zu beachten. Zuerst wird ESI zum Lesen verwendet und EDI zum Schreiben. Es ist einfach, sich an das zu erinnern, wenn man bedenkt, dass SI für *Source Index* und DI für *Destination Index* steht. Als nächstes muss man beachten, dass das Register, das die Daten hält, festgelegt ist (entweder AL, AX oder EAX). Schließlich muss man beachten, dass die speichernden Befehle ES benutzen, um das Segment zu bestimmen in das sie schreiben, nicht DS. In der protected Mode Programmierung ist dies gewöhnlich kein Problem, da es nur ein einziges Datensegment gibt und ES automatisch initialisiert sein sollte um sich auf dieses zu beziehen (genauso wie DS es ist). In der real Mode Programmierung jedoch, ist es für den Programmierer *sehr* wichtig, ES mit dem korrekten Segmentwert[4] zu initialisieren. Abbildung 5.8 zeigt ein Beispiel der Benutzung dieser Befehle, das einen Array in einen anderen kopiert.

[4]Eine weitere Komplikation ist, dass man, unter Benutzung eines einzelnen `MOV` Befehls, den Wert des DS Registers nicht direkt in das ES Register kopieren kann. Stattdessen muss der Wert von DS in ein Allzweckregister (wie AX) kopiert werden, um dann aus diesem Register nach ES kopiert zu werden, unter Verwendung von zwei `MOV` Befehlen.

Die Kombination eines `LODSx` mit einem `STOSx` Befehl (wie in Zeilen 13 und 14 von Abbildung 5.8) ist sehr verbreitet. Tatsächlich kann diese Kombination mit einem einzelnen `MOVSx` Stringbefehl durchgeführt werden. Abbildung 5.9 beschreibt die Operationen, die diese Befehle ausführen. Zeilen 13 und 14 von Abbildung 5.8 könnten mit dem gleichen Effekt durch einen einzelnen `MOVSD` Befehl ersetzt werden. Der einzige Unterschied wäre, dass das EAX Register in der Schleife überhaupt nicht verwendet werden würde.

5.2.2 Das `REP` Befehlspräfix

Die 80x86 Familie stellt ein spezielles Befehlspräfix[5], `REP` genannt, zur Verfügung, das mit den obigen Stringbefehlen verwendet werden kann. Dieses Präfix sagt der CPU, den nächsten Stringbefehl eine gegebene Anzahl mal zu wiederholen. Das ECX Register wird benutzt, um die Iterationen zu zählen (genauso wie bei einem `LOOP` Befehl). Unter Benutzung des `REP` Präfixes könnte die Schleife in Abbildung 5.8 (Zeilen 12 bis 15) durch eine einzelne Zeile ersetzt werden:

```
rep movsd
```

Abbildung 5.10 zeigt ein weiteres Beispiel, das den Inhalt eines Arrays löscht.

```
1  segment .bss
2  array   resd   10
3
4  segment .text
5          cld                     ; dies nicht vergessen!
6          mov    edi, array
7          mov    ecx, 10
8          xor    eax, eax
9          rep stosd
```

Abbildung 5.10: Beispiel einen Array zu löschen

5.2.3 Vergleichende Stringbefehle

Abbildung 5.11 zeigt verschiedene neue Stringbefehle, die verwendet werden können, um Speicher mit anderem Speicher oder einem Register zu vergleichen. Sie sind nützlich, um Arrays zu vergleichen oder zu durchsuchen. Sie setzen das FLAGS Register genauso wie der `CMP` Befehl. Die `CMPSx` Befehle vergleichen entsprechende Speicherstellen und die `SCASx` suchen Speicherstellen nach einem bestimmten Wert ab.

Abbildung 5.12 zeigt ein kurzes Codefragment, das die Zahl 12 in einem Doppelwortarray sucht. Der `SCASD` Befehl in Zeile 10 addiert immer 4 zu EDI, sogar wenn der gesuchte Wert gefunden wurde. Folglich, wenn man wünscht, die

[5]Ein Befehlspräfix ist kein Befehl, es ist ein spezielles Byte, das vor einen Stringbefehl gesetzt wird, um das Verhalten des Befehls zu modifizieren. Andere Präfixe werden auch benutzt, um die Segmentvoreinstellungen für die Speicherzugriffe zu überschreiben.

CMPSB	vergleicht Byte [DS:ESI] mit Byte [ES:EDI] ESI = ESI ± 1 EDI = EDI ± 1
CMPSW	vergleicht Word [DS:ESI] mit Word [ES:EDI] ESI = ESI ± 2 EDI = EDI ± 2
CMPSD	vergleicht Dword [DS:ESI] mit Dword [ES:EDI] ESI = ESI ± 4 EDI = EDI ± 4
SCASB	vergleicht AL mit [ES:EDI] EDI ± 1
SCASW	vergleicht AX mit [ES:EDI] EDI ± 2
SCASD	vergleicht EAX mit [ES:EDI] EDI ± 4

Abbildung 5.11: Vergleichende Stringbefehle

Adresse der im Array gefundenen 12 zu erhalten, ist es notwendig, 4 von EDI abzuziehen (wie in Zeile 16 getan).

5.2.4 Die REP*x* Befehlspräfixe

Es gibt verschiedene andere REP-ähnliche Befehlspräfixe, die mit den vergleichenden Stringbefehlen verwendet werden können. Abbildung 5.13 zeigt die beiden neuen Präfixe und beschreibt ihre Operationen. REPE und REPZ sind nur Synonyme für dasselbe Präfix (so wie REPNE und REPNZ). Wenn der wiederholte vergleichende Stringbefehl auf Grund des Vergleichs stoppt, wird das oder die Indexregister noch erhöht und ECX vermindert; jedoch hält das FLAGS Register noch den Zustand, der die Wiederholung beendete. So ist es möglich, das Z Flag zu benutzen, um festzustellen, ob die wiederholten Vergleiche auf Grund eines Vergleichs oder weil ECX Null wurde, beendet wurden.

Warum kann man nicht einfach nachsehen, ob ECX nach dem wiederholten Vergleich Null ist?

Abbildung 5.14 zeigt als Beispiel ein Codefragment, das bestimmt, ob zwei Speicherblöcke gleich sind. Das JE in Zeile 7 des Beispiels testet, um das Ergebnis des vorangehenden Befehls zu sehen. Wenn der wiederholte Vergleich anhielt, da er zwei ungleiche Bytes fand, wird das Z Flag immer noch gelöscht sein und es wird kein Sprung durchgeführt; wenn die Vergleiche jedoch anhielten, weil ECX Null wurde, wird das Z Flag immer noch gesetzt sein und der Code verzweigt zum `equal` Label.

5.2.5 Beispiel

Dieser Abschnitt enthält eine Assembler Quelldatei mit mehreren Funktionen, die Arrayoperationen mit den Stringbefehlen ausführen. Viele der Funktionen duplizieren bekannte C Bibliotheksfunktionen.

```
1  segment .bss
2  array resd  100
3
4  segment .text
5       cld
6       mov     edi, array       ; Zeiger zum Anfang des Arrays
7       mov     ecx, 100         ; Anzahl Elemente
8       mov     eax, 12          ; zu suchende Zahl
9  lp:
10      scasd
11      je      found
12      loop    lp
13      ; Code auszuführen, wenn nicht gefunden
14      jmp     onward
15 found:
16      sub     edi, 4           ; edi zeigt nun zur 12 im Array
17      ; Code auszuführen, wenn gefunden
18 onward:
```

Abbildung 5.12: Suchbeispiel

REPE, REPZ	wiederhole Befehl solange ZF gesetzt ist, aber höchstens ECX mal
REPNE, REPNZ	wiederhole Befehl solange ZF gelöscht ist, aber höchstens ECX mal

Abbildung 5.13: Die REP*x* Befehls-Präfixe

memory.asm

```
1  global _asm_copy, _asm_find, _asm_strlen, _asm_strcpy
2
3  segment .text
4  ; Funktion _asm_copy
5  ; kopiert einen Speicherblock
6  ; C Prototyp:
7  ; void asm_copy( void *dest, const void *src, unsigned sz );
8  ; Parameter:
9  ;   dest - Zeiger zum Ziel-Puffer
10 ;   src  - Zeiger zum Quell-Puffer
11 ;   sz   - Anzahl der zu kopierenden Bytes
12
13 ; als nächstes werden einige hilfreiche Symbole definiert
14
15 %define dest [ebp+8]
16 %define src  [ebp+12]
17 %define sz   [ebp+16]
18 _asm_copy:
19         enter   0, 0
20         push    esi
```

```
segment .text
     cld
     mov    esi, block1      ; Adresse des ersten Blocks
     mov    edi, block2      ; Adresse des zweiten Blocks
     mov    ecx, size        ; Größe der Blöcke in Byte
     repe   cmpsb            ; wiederhole, solange ZF gesetzt
     je     equal            ; Blöcke sind gleich, wenn ZF = 1
     ; Code ausführen, wenn Blöcke nicht gleich sind
     jmp    onward
equal:
     ; Code ausführen, wenn gleich
onward:
```

Abbildung 5.14: Speicherblöcke vergleichen

```
        push    edi

        mov     esi, src        ; esi = Adresse des Quell-Puffers
        mov     edi, dest       ; edi = Adresse des Ziel-Puffers
        mov     ecx, sz         ; ecx = Anzahl zu kopierender Bytes

        cld                     ; lösche Richtungsflag
        rep     movsb           ; führe movsb ECX mal aus

        pop     edi
        pop     esi
        leave
        ret


; Funktion _asm_find
; durchsucht Speicher nach einem gegebenen Byte
; void *asm_find( const void *src, char target, unsigned sz );
; Parameter:
;   src    - Zeiger zum zu durchsuchenden Puffer
;   target - zu suchender Bytewert
;   sz     - Anzahl der Bytes im Puffer
; Rückgabewert:
;   wenn target gefunden wird, wird der Zeiger zum ersten Auftreten
;     von target im Puffer zurückgegeben
;   sonst
;     wird NULL zurückgegeben
; Hinweis: target ist ein Bytewert, wird aber als Doppelwort auf den Stack geschoben.
;          Der Bytewert wird in den niederen 8 Bits gespeichert.
;


```

```
53  %define src    [ebp+8]
54  %define target [ebp+12]
55  %define sz     [ebp+16]
56
57  _asm_find:
58          enter   0, 0
59          push    edi
60
61          mov     eax, target     ; zu suchender Wert in al
62          mov     edi, src
63          mov     ecx, sz
64          cld
65
66          repne   scasb           ; scan bis ECX == 0 oder [ES:EDI] == AL
67
68          je      found_it        ; wenn ZF gesetzt, wurde Wert gefunden
69          mov     eax, 0          ; wenn nicht gefunden, gebe NULL zurück
70          jmp     short quit
71  found_it:
72          mov     eax, edi
73          dec     eax             ; wenn gefunden, gebe (EDI - 1) zurück
74  quit:
75          pop     edi
76          leave
77          ret
78
79  ; Funktion _asm_strlen
80  ; liefert die Größe eines Strings
81  ; unsigned asm_strlen( const char * );
82  ; Parameter:
83  ;   src - Zeiger zum String
84  ; Rückgabewert:
85  ;   Anzahl Zeichen im String (ohne 0 am Ende) (in EAX)
86
87  %define src [ebp+8]
88  _asm_strlen:
89          enter   0, 0
90          push    edi
91
92          mov     edi, src        ; edi = Zeiger zum String
93          mov     ecx, 0FFFFFFFFh ; benutze größtmögliches ECX
94          xor     al, al          ; al = 0
95          cld
96
97          repnz   scasb           ; durchsuche nach 0 Terminator
98
99  ;
100 ; repnz geht einen Schritt zu weit, deshalb ist Länge FFFFFFFE - ECX,
101 ; nicht FFFFFFFF - ECX
102 ;
```

```
103         mov     eax, 0FFFFFFFEh
104         sub     eax, ecx        ; length = 0FFFFFFFEh - ecx
105
106         pop     edi
107         leave
108         ret
109
110 ; Funktion _asm_strcpy
111 ; kopiert einen String
112 ; void asm_strcpy( char *dest, const char *src );
113 ; Parameter:
114 ;   dest - Zeiger zum Ziel-String
115 ;   src  - Zeiger zum Quell-String
116 ;
117 %define dest [ebp+8]
118 %define src  [ebp+12]
119 _asm_strcpy:
120         enter   0, 0
121         push    esi
122         push    edi
123
124         mov     edi, dest
125         mov     esi, src
126         cld
127 cpy_loop:
128         lodsb                   ; lade AL & inc ESI
129         stosb                   ; speichere AL & inc EDI
130         or      al, al          ; setze Bedingungsflags
131         jnz     cpy_loop        ; wenn nicht hinter 0 Terminator, weiter
132
133         pop     edi
134         pop     esi
135         leave
136         ret
```

memory.asm

memex.c

```
1  #include <stdio.h>
2
3  #define STR_SIZE 30
4  /* Prototypen */
5
6  void asm_copy( void *, const void *, unsigned ) __attribute__((cdecl));
7  void *asm_find( const void *,
8                  char target, unsigned ) __attribute__((cdecl));
9  unsigned asm_strlen( const char * ) __attribute__((cdecl));
10 void asm_strcpy( char *, const char * ) __attribute__((cdecl));
11
```

```
12  int main()
13  {
14    char st1[STR_SIZE] = "test string";
15    char st2[STR_SIZE];
16    char *st;
17    char  ch;
18
19    asm_copy(st2, st1, STR_SIZE); /* kopiere alle 30 Zeichen des Strings */
20     printf ("%s\n", st2);
21
22     printf ("Enter a char: "); /* suche nach Byte im String */
23    scanf("%c%*[^\n]", &ch);
24    st = asm_find(st2, ch, STR_SIZE);
25    if ( st )
26       printf ("Found it: %s\n", st);
27    else
28       printf ("Not found\n");
29
30    st1[0] = 0;
31     printf ("Enter string :");
32    scanf("%s", st1);
33     printf ("len = %u\n", asm_strlen(st1));
34
35    asm_strcpy(st2, st1); /* kopiere nur bedeutungsvolle Daten im String */
36     printf ("%s\n", st2);
37
38    return 0;
39  }
```

memex.c

Kapitel 6

Fließpunkt[1]

6.1 Fließpunkt-Darstellung

6.1.1 Nicht-ganzzahlige binäre Zahlen

Als im ersten Kapitel Zahlensysteme besprochen wurden, wurden nur ganzzahlige Werte betrachtet. Offensichtlich muss es möglich sein, nicht-ganzzahlige Zahlen genauso in anderen Basen zu repräsentieren wie in dezimal. In dezimal haben Ziffern rechts vom Dezimalpunkt zugeordnete negative Potenzen von Zehn:

$$0.123 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3}$$

Nicht überraschend, funktionieren binäre Zahlen ähnlich:

$$0.101_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.625$$

Diese Idee kann mit den ganzzahligen Methoden aus Kapitel 1 kombiniert werden, um eine allgemeine Zahl zu konvertieren:

$$110.011_2 = 4 + 2 + 0.25 + 0.125 = 6.375$$

Die Umwandlung von dezimal nach binär ist ebenfalls nicht sehr schwierig. Im Allgemeinen wird die dezimale Zahl in zwei Teile geteilt: Ganzzahl und Bruchteil. Den ganzzahligen Teil konvertiert man nach binär unter Verwendung der Methoden aus Kapitel 1. Der gebrochene Anteil wird unter Verwendung der unten beschriebenen Methode umgewandelt.

Betrachten wir einen binären Bruch mit den mit $a, b, c, \ldots$ bezeichneten Bits. Die Zahl sieht in binär dann so aus:

$$0.abcde\ldots$$

Multiplizieren wir die Zahl mit zwei. Die binäre Darstellung der neuen Zahl wird sein:

$$a.bcde\ldots$$

[1] Im Deutschen ist das Dezimaltrennzeichen das Komma. Programmiersprachen verwenden den im Englischen üblichen Dezimalpunkt. Um den Text mit den Beispielen und Programmen konsistent zu halten, ist vom Fließpunkt statt dem Fließkomma die Rede und wird der Punkt als Trennzeichen verwendet. [Anm. d. Ü.]

0.5625×2	$=$	1.125	erstes Bit	$=$	1
0.125×2	$=$	0.25	zweites Bit	$=$	0
0.25×2	$=$	0.5	drittes Bit	$=$	0
0.5×2	$=$	1.0	viertes Bit	$=$	1

Abbildung 6.1: Umwandlung von 0.5625 nach binär

Beachte, dass das erste Bit nun in der Einerstelle ist. Ersetzen wir a durch 0, bekommen wir:

$$0.bcde\ldots$$

und multiplizieren wieder mit zwei und erhalten:

$$b.cde\ldots$$

Nun ist das zweite Bit (b) in der Einerstelle. Dieses Verfahren kann so lange wiederholt werden, bis so viele Bits wie benötigt, gefunden wurden. Abbildung 6.1 zeigt ein wirkliches Beispiel, das 0.5625 ins Binäre umwandelt. Die Methode hält an, wenn der Bruchteil Null geworden ist.

0.85×2	$=$	1.7
0.7×2	$=$	1.4
0.4×2	$=$	0.8
0.8×2	$=$	1.6
0.6×2	$=$	1.2
0.2×2	$=$	0.4
0.4×2	$=$	0.8
0.8×2	$=$	1.6

Abbildung 6.2: Umwandlung von 0.85 nach binär

Als weiteres Beispiel betrachten wir die Konversion von 23.85 ins Binäre. Es ist einfach, den ganzzahligen Teil umzuwandeln ($23 = 10111_2$), aber was ist mit dem Bruchteil (0.85)? Abbildung 6.2 zeigt den Anfang dieser Berechnung. Wenn man sich die Zahlen sorgfältig ansieht, wird eine unendliche Schleife gefunden! Das bedeutet, dass 0.85 eine periodische Binärzahl ist (im Gegensatz zu einer periodischen Dezimalzahl in Basis 10)[2]. Es ist ein Muster in den Zahlen der Be-

[2]Es sollte nicht so überraschen, dass eine Zahl in einer Basis periodisch ist, aber nicht in einer anderen. Denken wir an $\frac{1}{3}$, es ist periodisch im Dezimalen, aber im Ternären (Basis 3) würde es 0.1_3 sein.

rechnung. Sieht man auf das Muster, kann man erkennen, dass $0.85 = 0.11\overline{0110}_2$. Folglich ist $23.85 = 10111.11\overline{0110}_2$.

Eine wichtige Konsequenz aus der obigen Berechnung ist, dass 23.85 unter Benutzung einer endlichen Anzahl von Bits binär nicht *exakt* repräsentiert werden kann. (Genauso wie $\frac{1}{3}$ in dezimal nicht mit einer endlichen Anzahl von Ziffern dargestellt werden kann.) Wie dieses Kapitel zeigt, werden in C `float` und `double` Variable binär gespeichert. Folglich können Werte wie 23.85 nicht exakt in diesen Variablen gespeichert werden. Nur eine Näherung von 23.85 kann gespeichert werden.

Um die Hardware zu vereinfachen, werden Fließpunktzahlen in einem konsistenten Format gespeichert. Dieses Format benutzt die wissenschaftliche Notation (aber in binär, unter Verwendung der Potenzen von zwei, nicht zehn). Zum Beispiel würde 23.85 oder $10111.11011001100110\ldots_2$ so gespeichert werden:

$$1.011111011001100110\ldots \times 2^{100}$$

(wobei der Exponent (100) in binär ist). Eine *normalisierte* Fließpunktzahl hat die Form:

$$1.sssssssssssssss \times 2^{eeeeeeee}$$

wobei $1.sssssssssssssss$ die *Signifikante* und $eeeeeeee$ der *Exponent* ist.

6.1.2 IEEE Fließpunkt Repräsentation

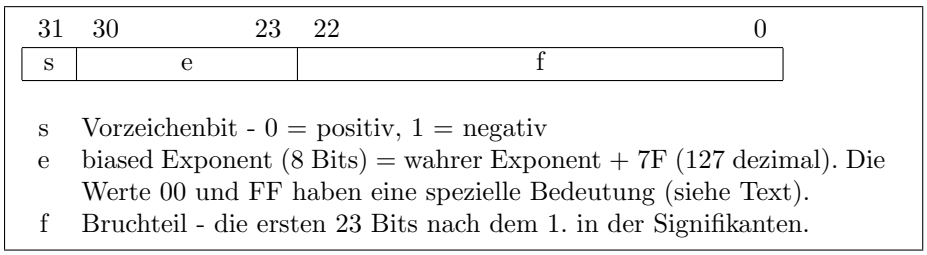
Die IEEE (Institute of Electrical and Electronic Engineers) ist eine internationale Organisation, die spezifische binäre Formate geschaffen hat, um Fließpunktzahlen zu speichern. Dieses Format wird auf den meisten (aber nicht allen!) Computern verwendet, die heute gefertigt werden. Oft wird sie durch die Hardware des Computers selbst unterstützt. Zum Beispiel benutzen es die numerischen (oder mathematischen) Coprozessoren von Intel (die in allen CPUs seit dem Pentium eingebaut sind). Die IEEE definiert zwei verschiedene Formate mit unterschiedlichen Genauigkeiten: einfache und doppelte Genauigkeit. Einfache Genauigkeit wird in C für `float` Variable benutzt und doppelte Genauigkeit wird für `double` Variable benutzt.

Intels mathematischer Coprozessor verwendet darüber hinaus eine dritte, höhere Genauigkeit, *extended precision* genannt. Tatsächlich sind alle Daten im Coprozessor selbst in dieser Genauigkeit. Wenn sie vom Coprozessor in den Speicher übertragen werden, werden sie automatisch entweder in die einfache oder doppelte Genauigkeit umgewandelt.[3] Extended precision verwendet ein gegenüber den IEEE Float- und Double-Formaten leicht unterschiedliches allgemeines Format und wird daher hier nicht besprochen.

IEEE Zahlen einfacher Genauigkeit

Fließpunkt einfache Genauigkeit verwendet 32 Bits um die Zahlen zu kodieren. Es ist gewöhnlich auf 7 signifikante dezimale Ziffern genau. Fließpunktzahlen werden in einem sehr viel komplizierteren Format gespeichert als Ganzzahlen.

[3] Die `long double` Typen einiger Compiler (wie Borland) verwenden diese erhöhte Genauigkeit. Jedoch benutzen andere Compiler die doppelte Genauigkeit sowohl für `double` wie auch für `long double`. (Das ist in ANSI C zulässig.)



s Vorzeichenbit - 0 = positiv, 1 = negativ
e biased Exponent (8 Bits) = wahrer Exponent + 7F (127 dezimal). Die Werte 00 und FF haben eine spezielle Bedeutung (siehe Text).
f Bruchteil - die ersten 23 Bits nach dem 1. in der Signifikanten.

Abbildung 6.3: IEEE single precision Format

Abbildung 6.3 zeigt das grundlegende Format einer IEEE Zahl einfacher Genauigkeit. Es gibt mehrere Eigenarten bei diesem Format. Fließpunktzahlen benutzen für negative Zahlen keine Repräsentation im Zweierkomplement. Sie benutzen eine signed magnitude Repräsentation. Bit 31 bestimmt das Vorzeichen der Zahl wie angegeben.

Der binäre Exponent wird nicht direkt gespeichert. Stattdessen wird die Summe des Exponenten und 7F in Bits 23 bis 30 gespeichert. Dieser *biased exponent* ist stets nicht-negativ.

Der gebrochene Anteil setzt eine normalisierte Signifikante (in der Form 1.*sssssssssssssss*) voraus. Da das erste Bit immer gesetzt ist, wird die führende Eins *nicht gespeichert!* Dies ermöglicht die Speicherung eines zusätzlichen Bits am Ende und erhöht so geringfügig die Genauigkeit. Diese Idee ist bekannt als die *hidden one representation*.

Wie würde 23.85 gespeichert werden? Zuerst ist es positiv, deshalb ist das Vorzeichenbit 0. Als nächstes ist der wahre Exponent 4, sodass der biased Exponent $7\mathrm{F} + 4 = 83_{16}$ ist. Schließlich ist der gebrochene Anteil 011111011001100 11001100 (denken wir daran, dass die führende Eins verborgen ist). Indem wir das alles zusammenfügen (zur Hilfe für die Erkennung der verschiedenen Abschnitte des Fließpunktformats wurden das Vorzeichenbit und der gebrochene Anteil unterstrichen und die Bits wurden in 4-bit Nibbles gruppiert):

$$\underline{0}\,100\ 0001\ 1\,\underline{011\ 1110\ 1100\ 1100\ 1100\ 1100}_2 = 41\mathrm{BECCCC}_{16}$$

Das ist nicht exakt 23.85 (da es ein periodischer Binärbruch ist). Wenn man das Obige zurück nach dezimal konvertiert, wird man finden, dass es ungefähr 23.849998474 ist. Diese Zahl ist sehr nahe an 23.85, aber es ist es nicht genau. In Wirklichkeit würde 23.85 in C nicht genau wie gezeigt repräsentiert werden. Da das am weitesten links stehende Bit, das von der exakten Repräsentation abgeschnitten wurde, 1 ist, wird das letzte Bit auf 1 aufgerundet. Deshalb würde 23.85 als 41 BE CC CD in hex in einfacher Genauigkeit repräsentiert werden. Dies nach dezimal gewandelt, gibt 23.850000381, das eine geringfügig bessere Approximation von 23.85 ist.

Man sollte immer im Gedächtnis behalten, dass die Bytes 41 BE CC CD auf verschiedene Arten interpretiert werden können, in Abhängigkeit davon, was ein Programm mit ihnen macht! Als eine einfachgenaue Fließpunktzahl repräsentieren sie 23.850000381, aber als ein Doppelwortinteger repräsentieren sie 1 103 023 309! Die CPU weiß nicht, welches die richtige Interpretation ist!

Wie würde −23.85 repräsentiert? Nur das Vorzeichenbit ändern: C1 BE CC CD. *Nicht* das Zweierkomplement nehmen!

Bestimmte Kombinationen von e und f haben spezielle Bedeutungen für IEEE Floats. Tabelle 6.1 beschreibt diese speziellen Werte. Unendlich wird durch einen Überlauf oder einer Division durch Null produziert. Ein undefiniertes Ergebnis wird produziert durch eine ungültige Operation, wie dem Versuch, die Quadratwurzel aus einer negativen Zahl zu ziehen, zwei Unendliche zu addieren, usw.

$e = 0$	und	$f = 0$	bezeichnet die Zahl Null (die nicht normalisiert werden kann). Beachte, dass es eine $+0$ und -0 gibt.
$e = 0$	und	$f \neq 0$	bezeichnet eine *denormalisierte Zahl*. Diese werden im nächsten Abschnitt besprochen.
$e = \text{FF}$	und	$f = 0$	bezeichnet unendlich (∞). Es gibt beides, positives und negatives Unendlich.
$e = \text{FF}$	und	$f \neq 0$	bezeichnet ein undefiniertes Ergebnis, als *NaN* (Not a Number) bekannt.

Tabelle 6.1: Spezielle Werte von f und e

Normalisierte Zahlen einfacher Genauigkeit reichen in ihrer Größe von 1.0×2^{-126} ($\approx 1.1755 \times 10^{-38}$) bis $1.11111\ldots \times 2^{127}$ ($\approx 3.4028 \times 10^{38}$).

Denormalisierte Zahlen

Denormalisierte Zahlen können mit Beträgen verwendet werden, die zu klein sind, um normalisiert zu werden (d. h. unter 1.0×2^{-126}). Betrachten wir zum Beispiel die Zahl $1.001_2 \times 2^{-129}$ ($\approx 1.6530 \times 10^{-39}$). In der gegebenen normalisierten Form ist der Exponent zu klein. Jedoch kann sie in der unnormalisierten Form repräsentiert werden: $0.01001_2 \times 2^{-127}$. Um diese Zahl zu speichern, wird der biased Exponent auf 0 gesetzt (siehe Tabelle 6.1) und der Bruchteil ist die vollständige Signifikante der Zahl, geschrieben als ein Produkt mit 2^{-127} (d. h. alle Bits werden gespeichert, einschließlich der Eins links des Dezimalpunkts). Die Repräsentation von 1.001×2^{-129} ist dann:

$$\underline{0}\,000\ 0000\ 0\,\underline{001\ 0010\ 0000\ 0000\ 0000\ 0000}$$

IEEE Zahlen doppelter Genauigkeit

IEEE doppelte Genauigkeit verwendet 64 Bits um Zahlen darzustellen und ist gewöhnlich bis auf ungefähr 15 signifikante Dezimalstellen genau. Wie Abbildung 6.4 zeigt, ist das grundlegende Format sehr ähnlich dem der einfachen Genauigkeit. Es werden mehr Bits für den biased Exponent (11) und den Bruchteil (52) verwendet als bei einfacher Genauigkeit.

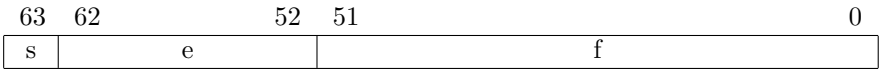


Abbildung 6.4: IEEE double precision Format

Der größere Bereich für den biased Exponent hat zwei Konsequenzen. Die erste ist, dass er als Summe des wahren Exponenten und 3FF (1023) berechnet wird (nicht 7F wie bei einfacher Genauigkeit). Zweitens ist ein großer Bereich von wahren Exponenten (und daher ein größerer Größenbereich) erlaubt. Größen doppelter Genauigkeit reichen von etwa 2.2251×10^{-308} bis 1.7977×10^{308}.

Es ist das größere Feld für den Bruchteil, das für das Anwachsen der Anzahl signifikanter Ziffern für Double-Werte verantwortlich ist.

Als ein Beispiel betrachten wir wieder 23.85. Der biased Exponent wird $4 + 3FF = 403$ in hex sein. Folglich würde die Double Repräsentation sein:

0 100 0000 0011 0111 1101 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010

oder 40 37 D9 99 99 99 99 9A in hex. Wenn man dies zurück nach dezimal konvertiert, findet man 23.8500000000000014 (es sind 12 Nullen!), was eine viel bessere Approximation von 23.85 ist.

Die doppelte Genauigkeit hat dieselben Spezialwerte wie bei einfacher Genauigkeit.[4] Denormalisierte Zahlen sind ebenfalls sehr ähnlich. Der einzige hauptsächliche Unterschied ist, dass unnormalisierte Double-Zahlen 2^{-1023} anstatt 2^{-127} benutzen.

6.2 Fließpunkt-Arithmetik

Fließpunktarithmetik auf einem Computer unterscheidet sich von der kontinuierlichen Mathematik. In der Mathematik können alle Zahlen als exakt betrachtet werden. Wie im vorhergehenden Abschnitt gezeigt, können auf einem Computer viele Zahlen, mit einer endlichen Anzahl von Bits, nicht exakt dargestellt werden. Alle Berechnungen werden mit einer begrenzten Genauigkeit durchgeführt. In den Beispielen dieses Abschnitts werden zur Einfachheit Zahlen mit einer 8-bit Signifikanten benutzt.

6.2.1 Addition

Um zwei Fließpunktzahlen zu addieren, müssen die Exponenten gleich sein. Wenn sie nicht schon gleich sind, müssen sie gleich gemacht werden, indem die Signifikante der Zahl mit dem kleineren Exponenten verschoben wird. Betrachten wir als Beispiel $10.375 + 6.34375 = 16.71875$ oder in binär:

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ + & 1.1001011 \times 2^2 \\ \hline \end{array}$$

Diese beiden Zahlen haben keine gleichen Exponenten, so verschieben wir die Signifikante um die Exponenten gleich zu machen und addieren dann:

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ + & 0.1100110 \times 2^3 \\ \hline & 10.0001100 \times 2^3 \end{array}$$

Beachte, dass das Verschieben von 1.1001011×2^2 die niederwertigste Eins wegfallen lässt und nach der Rundung 0.1100110×2^3 gibt. Das Ergebnis der Addition, 10.0001100×2^3 (oder 1.00001100×2^4) ist gleich 10000.110_2 oder 16.75. Das ist *nicht* gleich der exakten Antwort (16.71875)! Es ist nur eine Näherung, hervorgerufen durch die Rundungsfehler des Additionsprozesses.

Es ist wichtig, sich klar zu machen, dass Fließpunktarithmetik auf einem Computer (oder sonstigem Rechner) immer eine Näherung ist. Die Gesetze der Mathematik gelten mit Fließpunktzahlen auf einem Computer nicht immer. Die Mathematik setzt unendliche Genauigkeit voraus, der kein Computer entsprechen kann. Zum Beispiel lehrt die Mathematik, dass $(a + b) - b = a$ ist, jedoch muss das auf einem Computer nicht unbedingt exakt gelten!

[4] Der einzige Unterschied ist, dass für die Unendlich- und undefinierten Werte, der biased Exponent 7FF und nicht FF ist.

6.2.2 Subtraktion

Subtraktion arbeitet sehr ähnlich und hat die gleichen Probleme wie die Addition. Als ein Beispiel betrachten wir $16.75 - 15.9375 = 0.8125$:

$$\begin{array}{rr} & 1.0000110 \times 2^4 \\ - & 1.1111111 \times 2^3 \\ \hline \end{array}$$

Verschieben von 1.1111111×2^3 gibt (mit aufrunden) 1.0000000×2^4

$$\begin{array}{rr} & 1.0000110 \times 2^4 \\ - & 1.0000000 \times 2^4 \\ \hline & 0.0000110 \times 2^4 \end{array}$$

$0.0000110 \times 2^4 = 0.11_2 = 0.75$, das nicht exakt richtig ist.

6.2.3 Multiplikation und Division

Zur Multiplikation werden die Signifikanten multipliziert und die Exponenten werden addiert. Betrachten wir $10.375 \times 2.5 = 25.9375$:

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ \times & 1.0100000 \times 2^1 \\ \hline & 10100110 \\ + & 10100110 \\ \hline & 1.10011111000000 \times 2^4 \end{array}$$

Natürlich würde das wirkliche Ergebnis auf 8 Bits gerundet werden um zu geben:

$$1.1010000 \times 2^4 = 11010.000_2 = 26$$

Division ist komplizierter, hat aber ähnliche Problemen mit Rundungsfehlern.

6.2.4 Ableger für die Programmierung

Der Hauptpunkt dieses Abschnitts ist, dass Fließpunktrechnungen nicht exakt sind. Dem Programmierer muss dies bewusst sein. Ein verbreiteter Irrtum, den Programmierer mit Fließpunktzahlen machen, ist, sie unter der Annahme zu vergleichen, dass eine Berechnung exakt sei. Betrachten wir zum Beispiel eine Funktion `f(x)`, die eine komplexe Berechnung durchführt und ein Programm, das versucht, die Wurzeln der Funktion zu finden.[5] Man könnte versucht sein, das folgende Statement zu verwenden, um zu testen, ob `x` eine Wurzel ist:

```
if ( f(x) == 0.0 )
```

Aber was ist, wenn `f(x)` 1×10^{-30} zurückgibt? Es ist sehr wahrscheinlich, dass dies bedeutet, dass `x` eine *sehr* gute Näherung einer wahren Wurzel ist; jedoch wird der Vergleich falsch liefern. Es mag für `x` gar keinen IEEE Fließpunktwert geben, der genau Null zurückgibt, hervorgerufen durch die Rundungsfehler in `f(x)`.

[5] Eine Wurzel einer Funktion ist ein Wert x, derart, dass $f(x) = 0$ ist. [Nullstelle; Anm. d. Ü.]

Eine viel bessere Methode würde:

```
if ( fabs(f(x)) < EPS )
```

benutzen, wobei EPS ein Makro ist, das als ein sehr kleiner positiver Wert (wie 1×10^{-10}) definiert ist. Dies ist wahr, sobald f(x) sehr nahe Null ist. Im Allgemeinen benutzt man beim Vergleich eines Fließpunktwerts (sagen wir x) mit einem anderen (y):

```
if ( fabs((x − y) / y) < EPS )
```

6.3 Der numerische Coprozessor

6.3.1 Hardware

Die frühesten Intelprozessoren hatten keine Hardwareunterstützung für Fließpunktoperationen. Das bedeutet nicht, dass sie keine Fließpunktoperationen durchführen konnten. Es meint nur, dass sie von Prozeduren durchgeführt werden mussten, die aus vielen nicht-Fließpunktbefehlen zusammengesetzt waren. Für diese frühen Systeme lieferte Intel einen zusätzlichen Chip, der *mathematischer Coprozessor* genannt wurde. Ein mathematischer Coprozessor hat Maschinenbefehle, die viele Fließpunktoperationen viel schneller ausführen als bei der Benutzung von Softwareprozeduren (auf frühen Prozessoren wenigstens 10 Mal schneller!). Der Coprozessor für die 8086/8088 wurde 8087 genannt. Für die 80286 gab es einen 80287 und für die 80386 einen 80387. Der 80486DX Prozessor integrierte den mathematischen Coprozessor in die 80486 selbst.[6] Seit dem Pentium haben alle Generationen von 80x86 Prozessoren einen eingebauten mathematischen Coprozessor; er wird jedoch immer noch programmiert, als ob er eine getrennte Einheit wäre. Sogar frühere Systeme ohne einen Coprozessor können Software installieren, die einen mathematischen Coprozessor emuliert. Diese Emulationspakete werden automatisch aktiviert, wenn ein Programm einen Coprozessorbefehl ausführt und lassen eine Softwareprozedur laufen, die das gleiche Ergebnis liefert, wie es der Coprozessor getan hätte (obwohl natürlich viel langsamer).

Der numerische Coprozessor verfügt über acht Fließpunktregister. Jedes Register enthält 80 Datenbits. Fließpunktzahlen werden *immer* als 80-bit extended precision Zahlen in diesen Registern gespeichert. Die Register heißen `ST0`, `ST1`, `ST2`, ..., `ST7`. Die Fließpunktregister werden anders als die Integer-Register der Haupt-CPU benutzt. Die Fließpunktregister sind als *Stack* organisiert. Rufen wir uns in Erinnerung, dass ein Stack eine *Last-In First-Out* (LIFO) Liste ist. `ST0` bezieht sich immer auf den Wert an der Spitze des Stacks (TOS). Alle neuen Zahlen werden am TOS hinzugefügt. Existierende Zahlen wandern in den Stack hinein, um Platz für die neue Zahl zu machen.

Es gibt auch ein Statusregister im numerischen Coprozessor. Es enthält mehrere Flags. Es werden nur die 4 Flags, die für Vergleiche verwendet werden, besprochen: C_0, C_1, C_2 und C_3. Der Nutzen derselben wird später diskutiert.

[6] Jedoch hatte die 80486SX *keinen* integrierten Coprozessor. Es gab für diese Maschinen einen separaten 80487SX Chip.

6.3.2 Befehle

Um es einfach zu machen, die normalen CPU Befehle, von denen des Coprozessors zu unterscheiden, beginnen alle Mnemonics des Coprozessors mit einem `F`.

Laden und Speichern

Es gibt mehrere Befehle, die Daten auf die Spitze des Coprozessor Registerstacks laden:

`FLD` *source*	lädt eine Fließpunktzahl vom Speicher auf den TOS. *source* kann eine einfach, doppelt oder extended genaue Zahl oder ein Coprozessorregister sein.
`FILD` *source*	liest einen *Integer* aus dem Speicher, konvertiert ihn zu Fließpunkt und speichert das Ergebnis auf dem TOS. *source* kann entweder ein Wort, Doppelwort oder Quadwort sein.
`FLD1`	speichert eine Eins auf den TOS.
`FLDZ`	speichert eine Null auf den TOS.

Es gibt auch mehrere Befehle, die Daten vom Stack in den Speicher schreiben. Einige dieser Befehle führen auch eine *pop*-Operation aus, d. h. entfernen die Zahl vom Stack, während sie sie speichern.

`FST` *dest*	speichert den TOS (`ST0`) in den Speicher. *dest* kann entweder eine einfach oder doppelt genaue Zahl oder ein Coprozessorregister sein.
`FSTP` *dest*	speichert den TOS in den Speicher genau wie `FST`; jedoch wird der Wert, nachdem die Zahl gespeichert ist, vom Stack entfernt. *dest* kann entweder eine einfach, doppelt oder extended genaue Zahl oder ein Coprozessorregister sein.
code FIST *dest*	speichert den Wert im TOS, zu einem Integer gewandelt, in den Speicher. *dest* kann entweder ein Wort oder ein Doppelwort sein. Der Stack selbst bleibt unverändert. Wie die Fließpunktzahl in einen Integer gewandelt wird, hängt von einigen Bits im *Kontrollwort* des Coprozessors ab. Dies ist ein spezielles (nicht-Fließpunkt) Wortregister, das kontrolliert, wie der Coprozessor arbeitet. In der Grundeinstellung ist das Kontrollwort so initialisiert, dass er bei der Umwandlung in Integer zum nächsten Integer rundet. Jedoch können die Befehle `FSTCW` (Store Control Word) und `FLDCW` (Load Control Word) verwendet werden, um dieses Verhalten zu ändern.
`FISTP` *dest*	genau wie `FIST` mit Ausnahme zweier Dinge. Der Wert wird vom TOS entfernt und *dest* kann auch ein Quadwort sein.

```
1  segment .bss
2  array resq   SIZE
3  sum   resq   1
4
5  segment .text
6        mov    ecx, SIZE
7        mov    esi, array
8        fldz                    ; ST0 = 0
9  lp:
10       fadd   qword [esi]     ; ST0 += *(esi)
11       add    esi, 8          ; gehe zum nächsten Double
12       loop   lp
13       fstp   qword sum       ; speichere Ergebnis in sum
```

Abbildung 6.5: Beispiel einer Arraysummation

Es gibt zwei weitere Befehle, die Daten auf dem Stack selbst bewegen oder entfernen können.

`FXCH ST`*n*	vertauscht die Werte in `ST0` und `ST`*n* auf dem Stack (wobei *n* die Registernummer von 0 bis 7 ist).
`FFREE ST`*n*	gibt ein Register auf dem Stack frei, indem das Register als unbenutzt oder leer gekennzeichnet wird.

Addition und Subtraktion

Jeder der Additionsbefehle berechnet die Summe von `ST0` und einem weiteren Operanden. Das Ergebnis wird immer in einem Register des Coprozessors gespeichert.

`FADD` *src*	`ST0 +=` *src*. *src* kann jedes Coprozessorregister oder eine einfach oder doppelt genaue Zahl im Speicher sein.
`FADD` *dest*`, ST0`	*dest* `+= ST0`. *dest* kann jedes Coprozessorregister sein.
`FADDP` *dest* oder `FADDP` *dest*`, ST0`	*dest* `+= ST0`, dann wird der Wert vom TOS entfernt. *dest* kann jedes Coprozessorregister sein.
`FIADD` *src*	`ST0 += (float)` *src*. Addiert einen Integer zu `ST0`. *src* muss ein Wort oder Doppelwort im Speicher sein.

Es gibt doppelt so viele Subtraktionsbefehle wie Additionen, weil die Reihenfolge der Operanden bei der Subtraktion wichtig ist (d. h. $a+b=b+a$, aber $a-b \neq b-a$!). Zu jedem Befehle gibt es einen alternativen, der in der umgekehrten Anordnung subtrahiert. Diese umgekehrten Befehle enden alle entweder mit `R` oder `RP`. Abbildung 6.5 zeigt ein kurzes Codefragment, das die Elemente eines Double-Arrays aufsummiert. In Zeilen 10 und 13 muss man die Größe des Speicheroperanden angeben. Andernfalls würde der Assembler nicht wissen, ob der Operand ein Float (Doppelwort) oder ein Double (Quadwort) ist.

`FSUB` *src*	`ST0 -=` *src*. *src* kann jedes Coprozessorregister oder eine einfach oder doppelt genaue Zahl im Speicher sein.
`FSUBR` *src*	`ST0 =` *src* `- ST0`. *src* kann jedes Coprozessorregister oder eine einfach oder doppelte genaue Zahl im Speicher sein.
`FSUB` *dest*`, ST0`	*dest* `-= ST0`. *dest* kann jedes Coprozessorregister sein.
`FSUBR` *dest*`, ST0`	*dest* `= ST0 -` *dest*. *dest* kann jedes Coprozessorregister sein.
`FSUBP` *dest* oder `FSUBP` *dest*`, ST0`	*dest* `-= ST0`, dann wird der Wert vom TOS entfernt. *dest* kann jedes Coprozessorregister sein.
`FSUBRP` *dest* oder `FSUBRP` *dest*`, ST0`	*dest* `= ST0 -` *dest*, dann wird der Wert vom TOS entfernt. *dest* kann jedes Coprozessorregister sein.
`FISUB` *src*	`ST0 -= (float)` *src*. Zieht einen Integer von `ST0` ab. *src* muss ein Wort oder Doppelwort im Speicher sein.
`FISUBR` *src*	`ST0 = (float)` *src* `- ST0`. Zieht `ST0` von einem Integer ab. *src* muss ein Wort oder Doppelwort im Speicher sein.

Multiplikation und Division

Die Multiplikationsbefehle sind vollständig analog den Additionsbefehlen.

`FMUL` *src*	`ST0 *=` *src*. *src* kann jedes Coprozessorregister oder ein einfach oder doppelt genauer Wert im Speicher sein.
`FMUL` *dest*`, ST0`	*dest* `*= ST0`. *dest* kann jedes Coprozessorregister sein.
`FMULP` *dest* oder `FMULP` *dest*`, ST0`	*dest* `*= ST0`, dann wird der Wert vom TOS entfernt. *dest* kann jedes Coprozessorregister sein.
`FIMUL` *src*	`ST0 *= (float)` *src*. Multipliziert `ST0` mit einem Integer. *src* muss ein Wort oder Doppelwort im Speicher sein.

Nicht überraschend sind die Divisionsbefehle analog den Subtraktionsbefehlen. Division durch 0 führt zu Unendlich als Ergebnis.

`FDIV` *src*	`ST0 /=` *src*. *src* kann jedes Coprozessorregister oder eine einfach oder doppelt genaue Zahl im Speicher sein.
`FDIVR` *src*	`ST0 =` *src* `/ ST0`. *dest* kann jedes Coprozessorregister oder eine einfach oder doppelt genaue Zahl im Speicher sein.
`FDIV` *dest*`, ST0`	*dest* `/= ST0`. *dest* kann jedes Coprozessorregister sein.
`FDIVR` *dest*`, ST0`	*dest* `= ST0 /` *dest*. *dest* kann jedes Coprozessorregister sein.
`FDIVP` *dest* oder `FDIVP` *dest*`, ST0`	*dest* `/= ST0`, dann wird der Wert vom TOS entfernt. *dest* kann jedes Coprozessorregister sein.
`FDIVRP` *dest* oder `FDIVRP` *dest*`, ST0`	*dest* `= ST0 /` *dest*, dann wird der Wert vom TOS entfernt. *dest* kann jedes Coprozessorregister sein.
`FIDIV` *src*	`ST0 /= (float)` *src*. Dividiert `ST0` durch einen Integer. *src* muss ein Wort oder Doppelwort im Speicher sein.
`FIDIVR` *src*	`ST0 = (float)` *src* `/ ST0`. Dividiert einen Integer durch `ST0`. *src* muss ein Wort oder Doppelwort im Speicher sein.

Vergleiche

Der Coprozessor führt auch Vergleiche zwischen Fließpunktzahlen durch. Die `FCOM` Familie von Befehlen macht diese Operationen.

`FCOM` *src*	vergleicht `ST0` und *src*. *src* kann ein Coprozessorregister oder ein Float oder Double im Speicher sein.
`FCOMP` *src*	vergleicht `ST0` und *src*, dann wird der Wert vom TOS entfernt. *src* kann ein Coprozessorregister oder ein Float oder Double im Speicher sein.
`FCOMPP`	vergleicht `ST0` und `ST1`, dann werden zwei Werte vom TOS entfernt.
`FICOM` *src*	vergleicht `ST0` und `(float)` *src*. *src* kann ein Wort- oder Doppelwort-Integer im Speicher sein.
`FICOMP` *src*	vergleicht `ST0` und `(float)` *src*, dann wird der Wert vom TOS entfernt. *src* kann ein Wort- oder Doppelwort-Integer im Speicher sein.
`FTST`	vergleicht `ST0` mit 0.

Diese Befehle ändern die C_0, C_1, C_2 und C_3 Bits im Statusregister des Coprozessors. Unglücklicherweise ist es für die CPU nicht möglich, auf diese Bits direkt zuzugreifen. Die bedingten Vergleichsbefehle verwenden das FLAGS Register, nicht das Statusregister des Coprozessors. Jedoch ist es unter Benutzung einiger neuer Befehle relativ einfach, die Bits des Statuswortes in die entsprechenden Bits des FLAGS Registers zu übertragen:

`FSTSW` *dest*	Speichert das Statuswort des Coprozessors entweder in einem Wort im Speicher oder dem AX Register.
`SAHF`	Speichert das AH Register in das FLAGS Register.
`LAHF`	Lädt das AH Register mit den Bits des FLAGS Registers.

```
1  ;    if ( x > y )
2  ;
3       fld    qword [x]        ; ST0 = x
4       fcomp  qword [y]        ; vergleiche ST0 und y
5       fstsw  ax               ; kopiere C Bits nach FLAGS
6       sahf
7       jna    else_part        ; wenn x not above y, goto else_part
8  then_part:
9       ; Code für then Teil
10      jmp    end_if
11 else_part:
12      ; Code für else Teil
13 end_if:
```

Abbildung 6.6: Beispiel für Vergleiche

Abbildung 6.6 zeigt ein kurzes Beispielcodefragment. Zeilen 5 und 6 über-

tragen die Bits C_0, C_1, C_2 und C_3 vom Statuswort des Coprozessors in das FLAGS Register. Die Bits werden so übertragen, dass sie analog dem Ergebnis eines Vergleichs zweier *vorzeichenloser* Integer sind. Das ist der Grund, warum Zeile 7 einen `JNA` Befehl verwendet.

Der Pentium Pro (und spätere Prozessoren (Pentium II und III)) unterstützen zwei neue Vergleichsoperatoren, die direkt das FLAGS Register der CPU modifizieren.

`FCOMI` *src*	vergleicht `ST0` und *src*. *src* muss ein Coprozessorregister sein.
`FCOMIP` *src*	vergleicht `ST0` und *src*, dann wird der Wert vom TOS entfernt. *src* muss ein Coprozessorregister sein.

Abbildung 6.7 zeigt eine Beispielroutine, die das Maximum zweier Doubles unter Verwendung des `FCOMIP` Befehls findet. Verwechseln Sie diese Befehle nicht mit den Integervergleichsfunktionen (`FICOM` und `FICOMP`).

```
1  global _dmax
2
3  segment .text
4  ; Funktion _dmax
5  ; gibt das größere seiner beiden Double-Argumente zurück
6  ; C Prototyp:
7  ; double dmax( double d1, double d2 )
8  ; Parameter:
9  ;   d1   - erster Double
10 ;   d2   - zweiter Double
11 ; Rückgabewert:
12 ;   das größere von d1 und d2 (in ST0)
13 %define d1   ebp+8
14 %define d2   ebp+16
15 _dmax:
16      enter   0, 0
17
18      fld     qword [d2]
19      fld     qword [d1]      ; ST0 = d1, ST1 = d2
20      fcomip  st1             ; ST0 = d2
21      jna     short d2_bigger
22      fcomp   st0             ; hole d2 vom Stack
23      fld     qword [d1]      ; ST0 = d1
24      jmp     short exit
25 d2_bigger:                   ; d2 ist max, nichts zu tun
26 exit:
27      leave
28      ret
```

Abbildung 6.7: `FCOMIP` Beispiel

```
1  segment .data
2  x    dq     2.75            ; ins Double Format konvertiert
3  five dw     5
4
5  segment .text
6       fild   dword [five]     ; ST0 = 5
7       fld    qword [x]        ; ST0 = 2.75, ST1 = 5
8       fscale                  ; ST0 = 2.75 * 32, ST1 = 5
```

Abbildung 6.8: `FSCALE` Beispiel

Verschiedenartige Befehle

Dieser Abschnitt behandelt einige weitere verschiedenartige Befehle, die der Coprozessor bereitstellt.

`FCHS`	`ST0 = - ST0`. Ändert das Vorzeichen von `ST0`
`FABS`	`ST0` $= \lvert$`ST0`$\rvert$. Nimmt den absoluten Wert von `ST0`
`FSQRT`	`ST0` $= \sqrt{\texttt{ST0}}$. Zieht die Quadratwurzel aus `ST0`
`FSCALE`	`ST0` $=$ `ST0` $\times 2^{\lfloor \texttt{ST1} \rfloor}$. Multipliziert `ST0` schnell mit einer Potenz von zwei. `ST1` wird nicht vom Stack des Coprozessors entfernt. Abbildung 6.8 zeigt ein Beispiel, wie dieser Befehl eingesetzt wird.

6.4 Beispiele

6.4.1 Quadratische Formel

Das erste Beispiel zeigt, wie die quadratische Formel in Assembler kodiert werden kann. Erinnern wir uns, dass die quadratische Formel die Lösungen der quadratischen Gleichung berechnet:

$$ax^2 + bx + c = 0$$

Die Formel selbst liefert zwei Lösungen für x: x_1 und x_2.

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Der Ausdruck unter der Quadratwurzel ($b^2 - 4ac$) wird *Diskriminante* genannt. Ihr Wert ist nützlich bei der Bestimmung, welche der folgenden drei Möglichkeiten auf die Lösungen zutreffen.

1. Es gibt nur eine reelle degenerierte Lösung. $b^2 - 4ac = 0$
2. Es gibt zwei reelle Lösungen. $b^2 - 4ac > 0$
3. Es gibt zwei komplexe Lösungen. $b^2 - 4ac < 0$

Hier ist ein kleines C Programm, das die Assemblerroutine verwendet:

quadt.c

```
1  #include <stdio.h>
2
3  int quadratic( double, double, double, double *, double *);
4
5  int main()
6  {
7    double a, b, c, root1, root2;
8
9     printf ("Enter a, b, c: ");
10    scanf("%lf %lf %lf", &a, &b, &c);
11    if (quadratic( a, b, c, &root1, &root2) )
12       printf ("roots: %.10g %.10g\n", root1, root2);
13    else
14       printf ("No real roots\n");
15    return 0;
16 }
```

quadt.c

Hier ist die Assemblerroutine:

quad.asm

```
1  ; Funktion quadratic
2  ; findet die Lösungen der quadratischen Gleichung:
3  ;       a*x^2 + b*x + c = 0
4  ; C Prototyp:
5  ;   int quadratic( double a, double b, double c,
6  ;                  double *root1, double *root2 )
7  ; Parameter:
8  ;   a, b, c - Koeffizienten der Terme der quadratischen Gleichung (siehe oben)
9  ;   root1   - Zeiger auf Double um die erste Wurzel zu speichern
10 ;   root2   - Zeiger auf Double um die zweite Wurzel zu speichern
11 ; Rückgabewert:
12 ;   gibt 1 zurück wenn reelle Wurzeln gefunden, sonst 0
13
14 %define a               qword [ebp+8]
15 %define b               qword [ebp+16]
16 %define c               qword [ebp+24]
17 %define root1           dword [ebp+32]
18 %define root2           dword [ebp+36]
19 %define disc            qword [ebp-8]
20 %define one_over_2a     qword [ebp-16]
21
22 segment .data
23 MinusFour       dw      -4
24
25 segment .text
26         global  _quadratic
27 _quadratic:
```

```
28          push    ebp
29          mov     ebp, esp
30          sub     esp, 16         ; reserviere 2 Doubles (disc & one_over_2a)
31          push    ebx             ; muss originales ebx sichern
32
33          fild    word [MinusFour]; stack: -4
34          fld     a               ; stack: a, -4
35          fld     c               ; stack: c, a, -4
36          fmulp   st1             ; stack: a*c, -4
37          fmulp   st1             ; stack: -4*a*c
38          fld     b
39          fld     b               ; stack: b, b, -4*a*c
40          fmulp   st1             ; stack: b*b, -4*a*c
41          faddp   st1             ; stack: b*b - 4*a*c
42          ftst                    ; teste gegen 0
43          fstsw   ax
44          sahf
45          jb      no_real_solutions ; wenn disc < 0, keine reelle Lösung
46          fsqrt                   ; stack: sqrt(b*b - 4*a*c)
47          fstp    disc            ; speichere und pop Stack
48          fld1                    ; stack: 1.0
49          fld     a               ; stack: a, 1.0
50          fscale                  ; stack: a * 2^(1.0) = 2*a, 1
51          fdivp   st1             ; stack: 1/(2*a)
52          fst     one_over_2a     ; stack: 1/(2*a)
53          fld     b               ; stack: b, 1/(2*a)
54          fld     disc            ; stack: disc, b, 1/(2*a)
55          fsubrp  st1             ; stack: disc - b, 1/(2*a)
56          fmulp   st1             ; stack: (-b + disc)/(2*a)
57          mov     ebx, root1
58          fstp    qword [ebx]     ; speichere in *root1
59          fld     b               ; stack: b
60          fld     disc            ; stack: disc, b
61          fchs                    ; stack: -disc, b
62          fsubrp  st1             ; stack: -disc - b
63          fmul    one_over_2a     ; stack: (-b - disc)/(2*a)
64          mov     ebx, root2
65          fstp    qword [ebx]     ; speichere in *root2
66          mov     eax, 1          ; Rückgabewert ist 1
67          jmp     short quit
68
69  no_real_solutions:
70          mov     eax, 0          ; Rückgabewert ist 0
71
72  quit:
73          pop     ebx
74          mov     esp, ebp
75          pop     ebp
76          ret
```

quad.asm

6.4.2 Einen Array aus einer Datei lesen

In diesem Beispiel liest eine Assemblerroutine Doubles aus einer Datei. Hier ist ein kurzes C Testprogramm:

readt.c

```
1  /*
2   * Dieses Programm tested die 32-bit read_doubles() Assemblerprozedur.
3   * Es liest die Doubles von stdin. (Verwende Umleitung, um von Datei zu lesen.)
4   */
5  #include <stdio.h>
6  extern int read_doubles( FILE *, double *, int );
7  #define MAX 100
8
9  int main()
10 {
11   int i, n;
12   double a[MAX];
13
14   n = read_doubles(stdin, a, MAX);
15
16   for( i=0; i < n; i++ )
17     printf("%3d %g\n", i, a[i]);
18   return 0;
19 }
```

readt.c

Hier ist die Assemblerroutine:

read.asm

```
1  segment .data
2  format  db      "%lf", 0            ; Format für fscanf()
3
4  segment .text
5          global  _read_doubles
6          extern  _fscanf
7
8  %define SIZEOF_DOUBLE   8
9  %define FP              dword [ebp+8]
10 %define ARRAYP          dword [ebp+12]
11 %define ARRAY_SIZE      dword [ebp+16]
12 %define TEMP_DOUBLE     [ebp-8]
13
14 ;
15 ; Funktion _read_doubles
16 ; C Prototyp:
17 ;   int read_doubles( FILE *fp, double *arrayp, int array_size );
18 ; Diese Funktion liest Doubles aus einer Textdatei in einen Array,
19 ; bis EOF oder der Array voll ist.
```

```
20  ; Parameter:
21  ;   fp         - Datei-Zeiger (Quelle; muss für Input geöffnet sein)
22  ;   arrayp     - Zeiger zum Double-Array (Ziel)
23  ;   array_size - Anzahl der Elemente im Array
24  ; Rückgabewert:
25  ;   Anzahl der im Array gespeicherten Doubles (in EAX)
26
27  _read_doubles:
28          push    ebp
29          mov     ebp, esp
30          sub     esp, SIZEOF_DOUBLE  ; definiere einen Double auf dem Stack
31
32          push    esi                 ; sichere esi
33          mov     esi, ARRAYP         ; esi = ARRAYP
34          xor     edx, edx            ; edx = Array Index (anfänglich 0)
35
36  while_loop:
37          cmp     edx, ARRAY_SIZE     ; ist edx < ARRAY_SIZE ?
38          jnl     short quit          ; wenn nicht, beende Schleife
39  ;
40  ; rufe fscanf() auf um ein Double nach TEMP_DOUBLE zu lesen
41  ; fscanf() könnte edx ändern, so sichere es
42  ;
43          push    edx                 ; sichere edx
44          lea     eax, TEMP_DOUBLE
45          push    eax                 ; push &TEMP_DOUBLE
46          push    dword format        ; push &format
47          push    FP                  ; push Datei-Zeiger
48          call    _fscanf
49          add     esp, 12
50          pop     edx                 ; stelle edx wieder her
51          cmp     eax, 1              ; gab fscanf 1 zurück?
52          jne     short quit          ; wenn nicht, beende Schleife
53
54  ;
55  ; kopiere TEMP_DOUBLE nach ARRAYP[edx]
56  ; (Die 8 Bytes des Double werden durch zwei 4 Byte Kopien kopiert)
57  ;
58          mov     eax, [ebp-8]
59          mov     [esi + 8*edx], eax  ; zuerst kopiere die niedersten 4 Bytes
60          mov     eax, [ebp-4]
61          mov     [esi + 8*edx + 4], eax ; dann kopiere die höchsten 4 Bytes
62
63          inc     edx
64          jmp     while_loop
65
66  quit:
67          pop     esi                 ; stelle esi wieder her
68
69          mov     eax, edx            ; speichere Rückgabewert in eax
```

```
70
71          mov     esp, ebp
72          pop     ebp
73          ret
```

read.asm

6.4.3 Primzahlen finden

Dieses letzte Beispiel sieht sich das Auffinden von Primzahlen nochmals an. Diese Implementierung ist effizienter als die vorherige. Sie speichert die Primzahlen, die es gefunden hat in einem Array und dividiert nur durch die vorher gefundenen Primzahlen, anstatt durch jede ungerade Zahl, um neue Primzahlen zu finden.

Ein weiterer Unterschied ist, dass es die Quadratwurzel des Kandidaten für die nächste Primzahl berechnet, um zu bestimmen, an welchem Punkt es aufhören kann, nach Faktoren zu suchen. Es verändert das Kontrollwort des Coprozessors, sodass es, wenn es die Quadratwurzel als Integer speichert, abschneidet anstatt zu runden. Das wird durch Bits 10 und 11 des Kontrollworts kontrolliert. Diese Bits werden die RC (Rounding Control) Bits genannt. Wenn sie beide 0 sind (die Voreinstellung), rundet der Coprozessor, wenn er zu Integern konvertiert. Sind sie beide 1, schneidet der Coprozessor bei Integerwandlungen ab. Beachte, dass die Routine bedacht ist, das originale Kontrollwort zu speichern und wieder herzustellen, bevor sie zurückkehrt.

Hier ist das C Treiberprogramm:

fprime.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   /*
4    * Funktion find_primes
5    * findet die angegebene Anzahl von Primzahlen
6    * Parameter:
7    *    a – Array für Primzahlen
8    *    n – Anzahl zu findender Primzahlen
9    */
10  extern void find_primes( int *a, unsigned n );
11
12  int main()
13  {
14    int status;
15    unsigned i;
16    unsigned max;
17    int *a;
18
19    printf( "How many primes do you wish to find? " );
20    scanf( "%u", &max );
21
22    a = calloc( sizeof(int), max );
23
```

```
24      if ( a ) {
25
26        find_primes( a, max );
27
28        /* gebe die letzten 20 gefundenen Primzahlen aus */
29        for( i= ( max > 20 ) ? max - 20 : 0; i < max; i++ )
30          printf( "%3d %d\n", i+1, a[i] );
31
32        free( a );
33        status = 0;
34      }
35      else {
36         fprintf( stderr, "Can not create array of %u ints\n", max );
37        status = 1;
38      }
39
40      return status;
41    }
```

fprime.c

Hier ist die Assemblerroutine:

prime2.asm

```
1   segment .text
2   global  _find_primes
3   ;
4   ; Funktion find_primes
5   ; finde die angegebene Anzahl von Primzahlen
6   ; Parameter:
7   ;   array  - Array für die Primzahlen
8   ;   n_find - Anzahl zu findender Primzahlen
9   ; C Prototyp:
10  ;extern void find_primes( int *array, unsigned n_find )
11  ;
12  %define array         ebp+8
13  %define n_find        ebp+12
14  %define n             ebp-4         ; Anzahl bisher gefundener Primzahlen
15  %define isqrt         ebp-8         ; floor(sqrt(guess))
16  %define orig_cntl_wd  ebp-10        ; originales Kontrollwort
17  %define new_cntl_wd   ebp-12        ; neues Kontrollwort
18
19  _find_primes:
20          enter   12, 0               ; mache Platz für lokale Variable
21
22          push    ebx                 ; sichere mögliche Register-Variable
23          push    esi
24
25          fstcw   word [orig_cntl_wd] ; hole Kontrollwort
26          mov     ax, [orig_cntl_wd]
```

```
27          or      ax, 0C00h           ; setze Rundungsbits auf %11 (abschneiden)
28          mov     [new_cntl_wd], ax
29          fldcw   word [new_cntl_wd]
30
31          mov     esi, [array]        ; esi zeigt auf array
32          mov     dword [esi], 2      ; array[0] = 2
33          mov     dword [esi+4], 3    ; array[1] = 3
34          mov     ebx, 5              ; ebx = guess = 5
35          mov     dword [n], 2        ; n = 2
36  ;
37  ; Diese äußere Schleife findet bei jeder Iteration eine neue Primzahl,
38  ; welche sie dem Ende des Arrays hinzufügt. Anders als das frühere
39  ; Primzahl-Programm, bestimmt diese Funktion die Primalität nicht, indem
40  ; sie durch alle ungeraden Zahlen dividiert. Sie dividiert nur durch die
41  ; Primzahlen, die es bereits gefunden hat. (Das ist der Grund weshalb
42  ; sie in dem Array gespeichert werden.)
43  ;
44  while_limit:
45          mov     eax, [n]
46          cmp     eax, [n_find]       ; while ( n < n_find )
47          jnb     short quit_limit
48
49          mov     ecx, 1              ; ecx wird als Arrayindex benutzt
50          push    ebx                 ; speichere guess auf dem Stack
51          fild    dword [esp]         ; lade guess auf Coprozessor Stack
52          pop     ebx                 ; hole guess vom Stack
53          fsqrt                       ; finde sqrt(guess)
54          fistp   dword [isqrt]       ; isqrt = floor(sqrt(guess))
55  ;
56  ; Diese innere Schleife teilt guess (ebx) durch früher berechnete
57  ; Primzahlen bis es einen Primfaktor von guess findet (was bedeutet,
58  ; dass guess nicht prim ist) oder bis die zu dividierende Primzahl
59  ; größer als floor(sqrt(guess)) ist
60  ;
61  while_factor:
62          mov     eax, dword [esi + 4*ecx] ; eax = array[ecx]
63          cmp     eax, [isqrt]        ; while ( isqrt < array[ecx] ..
64          jnbe    short quit_factor_prime
65          mov     eax, ebx
66          xor     edx, edx
67          div     dword [esi + 4*ecx]
68          or      edx, edx            ; .. && guess % array[ecx] != 0 )
69          jz      short quit_factor_not_prime
70          inc     ecx                 ; versuche nächste Primzahl
71          jmp     short while_factor
72
73  ;
74  ; neue Primzahl gefunden !
75  ;
76  quit_factor_prime:
```

```
77          mov     eax, [n]
78          mov     dword [esi + 4*eax], ebx ; speichere guess am Arrayende
79          inc     eax
80          mov     [n], eax            ; inc n
81
82  quit_factor_not_prime:
83          add     ebx, 2              ; versuche nächste ungerade Zahl
84          jmp     short while_limit
85
86  quit_limit:
87
88          fldcw   word [orig_cntl_wd] ; stelle Kontrollwort wieder her
89          pop     esi                 ; stelle Register-Variable wieder her
90          pop     ebx
91
92          leave
93          ret
```

prime2.asm

Kapitel 7

Strukturen und C++

7.1 Strukturen

7.1.1 Einführung

Strukturen werden in C benutzt, um verwandte Daten zusammen in einer zusammengesetzten Variablen zu gruppieren. Diese Technik hat mehrere Vorteile:

1. Es lässt den Code klarer erscheinen, indem es zeigt, dass die in der Struktur definierten Daten eng miteinander verwandt sind.
2. Es vereinfacht, Daten an Funktionen zu übergeben. Anstatt mehrere Variable getrennt zu übergeben, können sie als eine einzige Einheit übergeben werden.
3. Es fördert die *Lokalität*[1] des Codes.

Vom Standpunkt des Assemblers aus kann eine Struktur als ein Array mit Elementen *variierender* Größe betrachtet werden. Die Elemente wirklicher Arrays haben immer gleiche Größe und Typ. Diese Eigenschaft ist es, die es einem erlaubt, die Adresse jeden Elements zu berechnen, wenn man die Startadresse des Arrays, die Größe der Elemente und den Index des verlangten Elements kennt.

Die Elemente einer Struktur müssen nicht von der gleichen Größe sein (und sind es gewöhnlich auch nicht). Aus diesem Grund muss jedes Element einer Struktur explizit spezifiziert werden und es erhält ein *Tag* (oder Namen) anstatt eines numerischen Indexes.

In Assembler wird auf das Element einer Struktur auf ähnlichem Weg zugegriffen wie auf ein Element eines Arrays. Um auf ein Element zuzugreifen, muss man die Startadresse der Struktur und den *relativen Offset* dieses Elements vom Anfang der Struktur kennen. Jedoch, anders als bei einem Array, wo der Offset aus dem Index des Elements berechnet werden kann, wird dem Element einer Struktur ein Offset durch den Compiler zugeordnet.

Betrachten wir zum Beispiel die folgende Struktur:

```
1  struct S {
2    short int x;     /* 2-Byte Integer */
```

[1] Siehe den Abschnitt über virtuelles Speichermanagement in jedem Lehrbuch über Betriebssysteme für eine Erklärung diesen Ausdrucks.

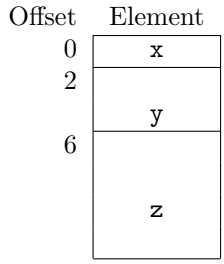


Abbildung 7.1: Struktur S

```
3     int     y;    /* 4-Byte Integer */
4     double  z;    /* 8-Byte Float   */
5   };
```

Abbildung 7.1 zeigt, wie eine Variable vom Typ `S` im Computerspeicher aussehen könnte. Der ANSI C Standard legt fest, dass die Elemente einer Struktur im Speicher in der gleichen Reihenfolge angeordnet sind wie sie in der Definition des `struct` definiert sind. Er legt ebenso fest, dass das erste Objekt ganz am Anfang der Struktur ist (d. h. Offset Null). Er definiert ebenfalls ein anderes hilfreiches Makro in der `stddef.h` Headerdatei namens `offsetof()`. Dieses Makro berechnet und gibt den Offset von irgendeinem Element einer Struktur zurück. Das Makro benötigt zwei Parameter, der erste ist der Name des *Typs* der Struktur, der zweite ist der Name des Elements, von dem der Offset zu finden ist. Deshalb würde `offsetof(S, y)` nach Abbildung 7.1 2 sein.

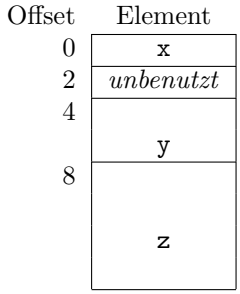


Abbildung 7.2: Struktur S

7.1.2 Speicherausrichtung

Erinnern wir uns, dass eine Adresse an einer Doppelwortgrenze ist, wenn sie durch 4 teilbar ist.

Wenn man das `offsetof` Makro verwendet, um den Offset von `y` unter Verwendung des *gcc* Compilers zu finden, wird man finden, dass es 4 zurückgibt, nicht 2! Warum? Weil *gcc* (und viele andere Compiler) Variable standardmäßig an Doppelwortgrenzen ausrichtet. Im 32-bit protected Mode liest die CPU Speicher schneller, wenn die Daten an einer Doppelwortgrenze beginnen. Abbildung 7.2 zeigt, wie die `S` Struktur unter Verwendung von *gcc* wirklich aussieht. Der Compiler fügt zwei unbenutzte Bytes in die Struktur ein, um `y` (und `z`) auf einer Doppelwortgrenze auszurichten. Dies zeigt, warum es eine gute Idee ist, `offsetof` zu benutzen, um die Offsets zu erhalten, anstatt sie selbst zu berechnen, wenn Strukturen benutzt werden, die in C definiert sind.

Natürlich, wenn die Struktur nur in Assembler benutzt wird, kann der Programmierer die Offsets selbst festlegen. Jedoch, wenn man C mit Assembler verwendet, ist es sehr wichtig, dass sowohl der Assemblercode als auch der C Code sich über die Offsets der Elemente einer Struktur einig sind! Eine Komplikation ist, dass unterschiedliche C Compiler den Elementen verschiedene Offsets geben können. Wie wir gesehen haben, kreïert zum Beispiel der *gcc* Compiler eine `S` Struktur, die wie in Abbildung 7.2 aussieht; jedoch würde Borlands Compiler eine Struktur erzeugen, die wie in Abbildung 7.1 aussieht. C Compiler liefern Wege, die für die Daten benutzte Ausrichtung festzulegen. Jedoch spezifiziert der ANSI C Standard nicht, wie dies getan werden soll und deshalb machen es verschiedene Compiler verschieden.

Der *gcc* Compiler besitzt eine flexible und komplizierte Methode, um die Ausrichtung zu spezifizieren. Der Compiler erlaubt einem, die Ausrichtung jeden Typs durch eine spezielle Syntax festzulegen. Zum Beispiel definiert die folgende Zeile:

```
typedef short int unaligned_int __attribute__((aligned(1)));
```

einen neuen Typ unter dem Namen `unaligned_int`, der auf Bytegrenzen ausgerichtet ist. (Ja, alle Klammern nach `__attribute__` sind erforderlich!) Die 1 im `aligned` Parameter kann durch andere Potenzen von zwei ersetzt werden, um andere Ausrichtungen zu spezifizieren (2 für Wortausrichtung, 4 für Doppelwortausrichtung, usw.) Wenn das Element `y` der Struktur geändert wurde um vom Typ `unaligned_int` zu sein, würde *gcc* `y` an den Offset 2 setzen. Jedoch würde `z` immer noch an Offset 8 sein, da Doubles per Voreinstellung auch auf Doppelwörtern ausgerichtet sind. Die Definition des Typs von `z` müsste ebenfalls geändert werden, um es auf Offset 6 zu setzen.

Der *gcc* Compiler erlaubt einem auch Strukturen zu *packen*. Das teilt dem Compiler mit, den kleinstmöglichen Platz für die Struktur zu verwenden. Abbildung 7.3 zeigt, wie `S` auf diese Art neu geschrieben werden könnte. Diese Form von `S` würde die kleinstmögliche Anzahl Bytes, 14, belegen.

```
1 struct S {
2   short int x;    /* 2-Byte Integer */
3   int       y;    /* 4-Byte Integer */
4   double    z;    /* 8-Byte Float   */
5 } __attribute__((packed));
```

Abbildung 7.3: Gepackte `struct` bei *gcc*

Microsofts und Borlands Compiler unterstützen beide die gleiche Methode die Ausrichtung festzulegen, durch Verwendung einer `#pragma` Direktive.

```
#pragma pack(1)
```

Die Direktive oben veranlasst den Compiler, die Elemente von Strukturen auf Bytegrenzen (d. h. ohne extra Einfügungen) zu packen. Die Eins kann durch zwei, vier, acht oder sechzehn ersetzt werden, um die Ausrichtung auf jeweils Wort-, Doppelwort-, Quadwort- und Paragraphen-Grenzen festzulegen. Die Direktive bleibt wirksam, bis sie durch eine andere Direktive überschrieben wird. Das kann Probleme verursachen, da diese Direktiven oft in Headerdateien ver-

```
1  #pragma pack(push)   /* sichere Zustand der Ausrichtung */
2  #pragma pack(1)      /* setze Ausrichtung auf Byte   */
3
4  struct S {
5    short int x;       /* 2-Byte Integer */
6    int       y;       /* 4-Byte Integer */
7    double    z;       /* 8-Byte Float   */
8  };
9
10 #pragma pack(pop)    /* stelle originale Ausrichtung wieder her */
```

Abbildung 7.4: Gepackte `struct` bei Microsoft oder Borland

```
1  struct S {
2    unsigned f1 : 3;   /*  3-bit Feld */
3    unsigned f2 : 10;  /* 10-bit Feld */
4    unsigned f3 : 11;  /* 11-bit Feld */
5    unsigned f4 : 8;   /*  8-bit Feld */
6  };
```

Abbildung 7.5: Bitfeld Beispiel

wendet werden. Wird die Headerdatei vor anderen Headerdateien mit Strukturen eingebunden, können diese Strukturen anders angelegt werden als sie durch Voreinstellung würden. Dies kann zu sehr schwer zu findenden Fehlern führen. Verschiedene Module eines Programms könnten die Elemente von Strukturen an *verschiedenen* Stellen anlegen!

Es gibt einen Weg, dieses Problem zu vermeiden. Microsoft und Borland unterstützen eine Methode, die gegenwärtige Ausrichtung zu speichern und später wiederherzustellen. Abbildung 7.4 zeigt, wie das gemacht werden würde.

7.1.3 Bitfelder

Bitfelder erlauben einem, Mitglieder eines `struct` zu spezifizieren, die nur eine spezifizierte Anzahl Bits benutzen. Die Anzahl der Bits muss kein Vielfaches von acht sein. Ein Mitglied eines Bitfelds wird wie ein **unsigned int** oder **int** Mitglied definiert, mit einem Doppelpunkt und der Bitgröße angehängt. Abbildung 7.5 zeigt ein Beispiel. Dies definiert eine 32 bit Variable, die in die folgenden Teile aufgeteilt ist:

8 Bits	11 Bits	10 Bits	3 Bits
f4	f3	f2	f1

Das erste Bitfeld ist den niederwertigsten Bits seines Doppelworts[2] zugeordnet.

Jedoch ist das Format nicht so einfach, wenn man sich ansieht, wie die Bits wirklich im Speicher abgelegt werden. Die Schwierigkeit tritt auf, wenn Bitfelder

[2] In Wirklichkeit gibt der ANSI/ISO C Standard dem Compiler einige Flexibilität, wie die Bits genau angelegt werden. Jedoch legen verbreitete C Compiler (*gcc*, *Microsoft* und *Borland*) die Felder so an.

Byte \ Bit	7	6	5	4	3	2	1	0
0	Operation Code (08h)							
1	Logical Unit #			msb of LBA				
2	middle of Logical Block Address							
3	lsb of Logicial Block Address							
4	Transfer Length							
5	Control							

Abbildung 7.6: SCSI Read Befehlsformat

Bytegrenzen überspannen, weil die Bytes auf einem little endian Prozessor im Speicher umgekehrt werden. Zum Beispiel sehen die Bitfelder der `S` Struktur im Speicher so aus:

5 Bits	3 Bits	3 Bits	5 Bits	8 Bits	8 Bits
f2l	f1	f3l	f2m	f3m	f4

Das *f2l* Label bezieht sich auf die letzten fünf Bits (d. h. die fünf niederwertigsten Bits) des *f2* Bitfeldes. Das *f2m* Label bezieht sich auf die fünf höchstwertigen Bits von *f2*. Die doppelten senkrechten Linien zeigen die Bytegrenzen. Wenn man all die Bytes umdreht, werden die Teile der *f2* und *f3* Felder an der richtigen Stelle wieder vereinigt.

Die physikalische Speicherbelegung ist gewöhnlich nicht wichtig, bis die Daten in oder aus dem Programm übertragen werden (was eigentlich bei Bitfeldern ziemlich verbreitet ist). Für Hardware Geräteschnittstellen ist es verbreitet, ungewöhnliche Anzahlen von Bits zu verwenden, sodass Bitfelder zu ihrer Repräsentation nützlich sein könnten.

Ein Beispiel ist SCSI[3]. Ein direktes Lesekommando für ein SCSI-Gerät wird spezifiziert, indem eine sechs Byte Nachricht im in Abbildung 7.6 spezifizierten Format an das Gerät gesendet wird. Die Schwierigkeit, dies mit Bitfeldern zu repräsentieren, macht die *logcial block address*, die 3 verschiedene Bytes des Kommandos überspannt. Aus Abbildung 7.6 sieht man, dass die Daten im big endian Format gespeichert werden. Abbildung 7.7 zeigt eine Definition, die versucht, mit allen Compilern zu funktionieren. Die ersten beiden Zeilen definieren ein Makro, das wahr ist, wenn der Code mit Microsofts oder Borlands Compiler übersetzt wird. Die potenziell verwirrenden Teile sind Zeilen 11 bis 14. Zuerst könnte man sich wundern, warum die lba_mid und lba_lsb Felder getrennt definiert wurden und nicht als ein einzelnes 16-bit Feld? Der Grund ist, dass die Daten in big endian Ordnung sind. Ein 16-bit Feld würde durch den Compiler in little endian Ordnung gespeichert werden. Als nächstes erscheinen die lba_msb und logical_unit Felder vertauscht zu sein; jedoch ist dies nicht der Fall. Sie müssen in dieser Reihenfolge angelegt werden. Abbildung 7.8 zeigt, wie die Felder als eine 48-bit Einheit angelegt werden. (Die Bytegrenzen sind wieder durch doppelte Linien bezeichnet.) Wenn dies im Speicher in little endian Ordnung gespeichert wird, werden die Bits im gewünschten Format (Abbildung 7.6) angeordnet.

Um die Sache noch komplizierter zu machen, arbeitet die Definition für das SCSI_read_cmd nicht ganz korrekt mit Microsofts C Compiler. Wird der

[3]Small Computer System Interface, ein Industriestandard für Festplatten, usw.

```
1  #define MS_OR_BORLAND (defined(__BORLANDC__) \
2                         || defined(_MSC_VER))
3
4  #if MS_OR_BORLAND
5  # pragma pack(push)
6  # pragma pack(1)
7  #endif
8
9  struct SCSI_read_cmd {
10   unsigned opcode : 8;
11   unsigned lba_msb : 5;
12   unsigned logical_unit : 3;
13   unsigned lba_mid : 8;    /* mittlere Bits */
14   unsigned lba_lsb : 8;
15   unsigned transfer_length : 8;
16   unsigned control : 8;
17 }
18 #if defined(__GNUC__)
19     __attribute__((packed))
20 #endif
21 ;
22
23 #if MS_OR_BORLAND
24 # pragma pack(pop)
25 #endif
```

Abbildung 7.7: SCSI Read Command Format Struktur

8 Bits	3 Bits	5 Bits	8 Bits	8 Bits	8 Bits	8 Bits
opcode	logical_unit	lba_msb	lba_mid	lba_lsb	transfer_length	control

Abbildung 7.8: Aufteilung der `SCSI_read_cmd` Felder

sizeof(SCSI_read_cmd) Ausdruck ausgewertet, gibt Microsoft C 8 zurück, nicht 6! Das ist, weil der Microsoft Compiler den Typ des Bitfeldes bei der Festlegung, wie die Bits angelegt werden, benutzt. Da alle Bitfelder als vom Typ **unsigned** definiert sind, fügt der Compiler am Ende der Struktur zwei Bytes ein, um sie auf eine ganzzahligen Zahl von Doppelwörtern zu bringen. Das kann kuriert werden, indem stattdessen alle Felder **unsigned short** gemacht werden. Jetzt braucht der Microsoft Compiler keine Bytes hinzuzufügen, da sechs Bytes eine ganzzahliges Anzahl von zwei-Byte Wörtern sind.[4] Die anderen Compiler arbeiten mit dieser Änderung ebenfalls korrekt. Abbildung 7.9 zeigt noch eine weitere Definition, die mit allen drei Compilern funktioniert. Es vermeidet alle, bis auf zwei der Bitfelder, durch Verwendung von **unsigned char**.

Der Leser sollte nicht entmutigt sein, wenn er die vorhergehende Diskussion verwirrend fand. Sie ist verwirrend! Der Autor findet es oft weniger verwirrend,

[4]Die Mischung verschiedener Typen von Bitfeldern führt zu sehr verwirrendem Verhalten. Der Leser ist zum Experimentieren eingeladen.

```
1  struct SCSI_read_cmd {
2    unsigned char opcode;
3    unsigned char lba_msb : 5;
4    unsigned char logical_unit : 3;
5    unsigned char lba_mid;    /* mittlere Bits */
6    unsigned char lba_lsb;
7    unsigned char transfer_length;
8    unsigned char control;
9  }
10 #if defined(__GNUC__)
11     __attribute__((packed))
12 #endif
13 ;
```

Abbildung 7.9: Alternative SCSI Read Command Format Struktur

die Bitfelder überhaupt zu vermeiden und Bitoperationen zu verwenden, um die Bits von Hand zu untersuchen und zu modifizieren.

7.1.4 Strukturen in Assembler benutzen

Wie oben diskutiert, ist der Zugriff auf eine Struktur in Assembler fast ganz so, wie der Zugriff auf einen Array. Für ein einfaches Beispiel betrachten wir, wie man eine Assemblerroutine schreiben würde, die das Element `y` einer Struktur `S` löschen würde. Unter der Annahme, dass der Prototyp der Routine:

```
void zero_y( S *s_p );
```

ist, würde die Assemblerroutine so sein:

```
1  %define    y_offset  4
2  _zero_y:
3      enter  0, 0
4      mov    eax, [ebp+8]      ; hole s_p (struct-Zeiger) vom Stack
5      mov    dword [eax+y_offset], 0
6      leave
7      ret
```

C erlaubt einem, eine Struktur an eine Funktion per Wert zu übergeben; jedoch ist das fast immer eine schlechte Idee. Wenn als Wert übergeben, müssen die gesamten Daten der Struktur auf den Stack kopiert und durch die Routine zurückgeholt werden. Es ist sehr viel effizienter, stattdessen einen Zeiger auf die Struktur zu übergeben.

C erlaubt auch einen Strukturtyp als Rückgabewert einer Funktion zu verwenden. Offensichtlich kann eine Struktur nicht im `EAX` Register zurückgegeben werden. Verschiedene C Compiler behandeln diese Situation verschieden. Eine verbreitete Lösung, die Compiler verwenden, ist, intern die Funktion umzuschreiben zu einer, die einen Strukturzeiger als Parameter hat. Der Zeiger wird

benutzt, um den Rückgabewert in eine außerhalb der aufgerufenen Routine definierte Struktur abzulegen.

Die meisten Assembler (einschließlich NASM) besitzen eine eingebaute Unterstützung, um Strukturen in Ihrem Assemblercode zu definieren. Für Einzelheiten, ziehen Sie Ihre Dokumentation zu Rate.

7.2 Assembler und C++

Die Programmiersprache C++ ist eine Erweiterung der Sprache C. Viele der grundlegenden Regeln, um C mit Assembler zu verbinden, treffen auch auf C++ zu. Jedoch müssen einige Regeln modifiziert werden. Ebenso sind einige der Erweiterungen von C++ mit der Kenntnis von Assemblersprache leichter zu verstehen. Dieser Abschnitt setzt eine elementare Kenntnis von C++ voraus.

7.2.1 Überladung und Dekoration[5] von Namen

C++ erlaubt, dass verschiedene Funktionen (und Mitgliedsfunktionen von Klassen) mit dem gleichen Namen definiert werden. Wenn sich mehr als eine Funktion den gleichen Namen teilen, sagt man, die Funktionen sind *überladen*. Wenn in C zwei Funktionen mit demselben Namen definiert werden, wird der Linker einen Fehler generieren, weil er in der Objektdatei, die er verbindet, zwei Definitionen für das gleiche Symbol findet. Betrachten wir zum Beispiel den Code in Abbildung 7.10. Der äquivalente Assemblercode würde zwei Labels mit Namen `_f` definieren, was offensichtlich ein Fehler ist.

```
1  #include <stdio.h>
2
3  void f( int x )
4  {
5     printf ("%d\n", x);
6  }
7
8  void f( double x )
9  {
10    printf ("%g\n", x);
11 }
```

Abbildung 7.10: Zwei `f()` Funktionen

C++ verwendet den gleichen Linkprozess wie C, aber vermeidet diesen Fehler, indem es *Namensdekoration* oder Modifikation der Symbole durchführt, die benutzt wird, um die Funktion mit einem Label zu versehen. Auf eine Weise verwendet auch schon C Namensdekoration. Es fügt einen Unterstrich an den Namen der C Funktion an, wenn es das Label für die Funktion generiert. Jedoch wird C die Namen beider Funktionen in Abbildung 7.10 in der gleichen

[5] Das im Original verwendete *name mangling* wird auch als *name decoration* bezeichnet. In der Übersetzung wurde der Begriff ‚Dekoration' einem ‚in die Mangel genommenen' oder gar ‚verstümmelten' Namen vorgezogen. [Anm. d. Ü.]

Weise dekorieren und einen Fehler produzieren. C++ benutzt einen fortschrittlicheren Dekorationsprozess, der zwei verschiedene Labels für die Funktionen liefert. Zum Beispiel würde der ersten Funktion in Abbildung 7.10 durch DJGPP das Label `_f__Fi` und der zweiten Funktion `_f__Fd` zugeordnet werden. Dies vermeidet jegliche Linkfehler.

Unglücklicherweise gibt es keinen Standard wie Namen in C++ behandelt werden und verschiedene Compiler dekorieren Namen verschieden. Zum Beispiel würde Borlands C++ die Labels `@f$qi` und `@f$qd` für die zwei Funktionen in Abbildung 7.10 vergeben. Jedoch sind die Regeln nicht völlig zufällig. Der dekorierte Name kodiert die *Signatur* der Funktion. Die Signatur einer Funktion wird durch die Reihenfolge und den Typ ihrer Parameter definiert. Beachte, dass die Funktion, die ein einzelnes `int` Argument besitzt, ein *i* am Ende ihres dekorierten Namens (für DJGPP und Borland) hat und diejenige, die ein `double` Argument besitzt, hat ein *d* am Ende ihres dekorierten Namens. Wenn es eine Funktion namens `f` mit dem Prototypen:

```
void f( int x, int y, double z );
```

gäbe, würde DJGPP ihren Namen zu `_f__Fiid` verändern und Borland würde `@f$qiid` daraus machen.

Der Rückgabetyp einer Funktion ist *kein* Bestandteil der Signatur der Funktion und wird in ihrem dekorierten Namen nicht kodiert. Diese Tatsache erklärt eine Regel der Überladung in C++. Nur Funktionen, deren Signaturen eindeutig sind, können überladen werden. Wie man sehen kann, wenn in C++ zwei Funktionen mit gleichem Namen und Signatur definiert werden, werden sie den gleichen dekorierten Namen bekommen und werden einen Linkerfehler hervorrufen. In der Voreinstellung werden die Namen aller C++ Funktionen dekoriert, selbst die, die nicht überladen sind. Wenn er eine Datei kompiliert, hat der Compiler keine Möglichkeit zu wissen, ob eine bestimmte Funktion überladen ist oder nicht und daher dekoriert er alle Namen. Tatsächlich modifiziert er ebenso die Namen globaler Variablen, indem er die Typen der Variablen auf ähnliche Art wie Funktionssignaturen kodiert. Wenn man folglich eine globale Variable in einer Datei als von einem bestimmten Typ definiert und versucht, sie in einer anderen Datei mit einem falschen Typ zu benutzen, dann wird ein Linkerfehler auftreten. Dieses Charakteristikum von C++ ist als *typsicheres Linken* bekannt. Es deckt ebenso einen anderen Typ von Fehlern auf, inkonsistente Prototypen. Dieser tritt auf, wenn die Definition einer Funktion in einem Modul nicht mit dem in einem anderen Modul verwendeten Prototypen übereinstimmt. In C kann dies ein sehr schwer zu debuggendes Problem sein. C fängt diesen Fehler nicht ab. Das Programm wird übersetzt und gelinkt, aber wird ein undefiniertes Verhalten zeigen, da der rufende Code andere Typen auf dem Stack ablegen wird als die Funktion erwartet. In C++ wird es einen Linkerfehler hervorrufen.

Wenn der C++ Compiler einen Funktionsaufruf analysiert, schaut er nach einer übereinstimmenden Funktion, indem er nach den Typen der an die Funktion übergebenen Argumente schaut.[6] Wenn er eine Übereinstimmung findet, erzeugt er einen `CALL` zur korrekten Funktion, indem er die Regeln der Namensdekoration des Compilers anwendet.

[6]Der Treffer muss keine exakte Übereinstimmung sein, der Compiler wird Treffer berücksichtigen, die durch Casts der Argumente entstehen. Die Regeln für diesen Prozess gehen über den Rahmen dieses Buches hinaus. Für Einzelheiten ziehe man ein C++ Buch zu Rate.

Da verschiedene Compiler verschiedene Namensdekorationsregeln benutzen, kann es sein, dass durch verschiedene Compiler übersetzter C++ Code nicht zusammen gelinkt werden kann. Diese Tatsache ist wichtig, wenn die Benutzung einer vorkompilierten C++ Bibliothek in Betracht gezogen wird! Wenn man in Assembler eine Funktion schreiben möchte, die zusammen mit C++ Code benutzt wird, muss man die Namensdekorationsregeln für den benutzten C++ Compiler kennen (oder die unten erklärte Technik anwenden).

Der scharfsinnige Leser mag sich fragen, ob der Code in Abbildung 7.10 wie erwartet arbeiten wird. Da C++ alle Funktionen dekoriert, wird die `printf` Funktion dekoriert und der Compiler wird keinen `CALL` zum Label `_printf` produzieren. Das ist ein stichhaltiger Einwand! Wenn der Prototyp für `printf` einfach an den Anfang der Datei gestellt wird, würde dies geschehen. Der Prototyp ist:

```
int printf( const char *, ... );
```

DJGPP würde das zu `_printf__FPCce` dekorieren. (Das `F` steht für *Funktion*, `P` für *Pointer*, `C` für *Const*, `c` für *Char* und `e` für *Ellipse*.) Das würde nicht die reguläre C Bibliotheksfunktion `printf` aufrufen! Natürlich muss es für C++ Code einen Weg geben um C Code aufzurufen. Dies ist sehr wichtig, weil es *eine Menge* von nützlichem alten C Code gibt. Zusätzlich, dass es einem erlaubt, bestehenden C Code aufzurufen, erlaubt einem C++ auch Assemblercode, unter Verwendung der normalen C Dekorations-Konvention, aufzurufen.

C++ erweitert das `extern` Schlüsselwort, um ihm spezifizieren zu können, dass die Funktion oder globale Variable, die es modifiziert, normale C Konvention benutzt. In der Terminologie von C++ benutzt die Funktion oder globale Variable *C linkage*. Um `printf` zum Beispiel mit C Bindung zu deklarieren, benutzt man den Prototypen:

```
extern "C" int printf( const char *, ... );
```

Das instruiert den Compiler, für diese Funktion nicht die C++ Dekorationsregeln zu verwenden, sondern stattdessen die C Regeln anzuwenden. Dadurch kann jedoch die `printf` Funktion nicht mehr überladen werden. Das stellt den einfachsten Weg dar, um C++ und Assembler zu verbinden, indem man die Funktion so definiert, dass sie C Bindung verwendet und dann die Aufrufkonvention von C benutzt.

Zur Bequemlichkeit erlaubt C++ auch, die Bindung eines Blocks von Funktionen und globalen Variablen zu definieren. Der Block wird durch die üblichen geschweiften Klammern eingeschlossen.

```
extern "C" {
  /* globale Variable and Funktions-Prototypen in C Bindung */
}
```

Wenn man die ANSI C Headerdateien, die heute mit C/C++ Compilern kommen, untersucht, wird man das Folgende nahe dem Anfang jeder Headerdatei finden:

```
#ifdef __cplusplus
extern "C" {
#endif
```

```
1  void f( int &x )      // das & bezeichnet einen Referenz−Parameter
2  { x++; }
3
4  int main()
5  {
6    int y = 5;
7    f(y);               // Referenz auf y wird übergeben, beachte kein & hier!
8     printf ("%d\n", y); // gibt 6 aus!
9    return 0;
10 }
```

Abbildung 7.11: Beispiel zu Referenzen

Und ein ähnliches Konstrukt nahe dem Ende, das eine schließende geschweifte Klammer enthält. C++ Compiler definieren das `__cplusplus` Makro (mit *zwei* führenden Unterstrichen). Das obige Fragment schließt die gesamte Headerdatei in einen `extern "C"` Block ein, wenn die Headerdatei als C++ kompiliert wird, aber macht nichts, wenn als C kompiliert (da ein C Compiler einen Syntaxfehler für `extern "C"` liefern würde). Die gleiche Technik kann von jedem Programmierer benutzt werden, um eine Headerdatei für Assemblerroutinen zu schaffen, die sowohl mit C als auch mit C++ benutzt werden kann.

7.2.2 Referenzen

Referenzen sind eine andere neue Eigenschaft von C++. Sie erlauben einem, Parameter an Funktionen zu übergeben, ohne explizit Zeiger zu verwenden. Betrachten wir zum Beispiel den Code in Abbildung 7.11. Tatsächlich sind Referenzparameter ziemlich einfach, sie sind wirklich nur Zeiger. Der Compiler verbirgt dies nur vor dem Programmierer (genau wie Pascal Compiler `var` Parameter als Zeiger implementieren). Wenn der Compiler für den Funktionsaufruf in Zeile 7 Assemblercode generiert, übergibt er nur die *Adresse* von `y`. Wenn man Funktion `f` in Assembler schreibt, würde sie sich verhalten, als ob der Prototyp wäre:[7]

```
void f( int *xp );
```

Referenzen sind nur eine Annehmlichkeit, die speziell für Überladungen von Operatoren nützlich sind. Das ist eine weitere Eigenschaft von C++, die einem erlaubt, für allgemeine Operatoren eine Bedeutung für Strukturen oder Klassentypen zu definieren. Zum Beispiel ist es ein allgemeiner Gebrauch, den Plus (`+`) Operator zur Verkettung von Stringobjekten zu definieren. So, wenn `a` und `b` Strings wären, würde `a + b` die Verkettung der Strings `a` und `b` liefern. C++ würde eigentlich eine Funktion aufrufen, um dies zu tun (tatsächlich könnte dieser Ausdruck in Funktions-Notation als `operator +(a, b)` umgeschrieben werden). Zur Effizienz würde man gerne die Adressen der Stringobjekte übergeben, anstatt sie als Werte zu übergeben. Ohne Referenzen könnte dies als `operator +(&a, &b)` geschrieben werden, aber das würde erfordern, dass man

[7] Natürlich könnte man die Funktion mit C Bindung definieren wollen, um Namensdekoration, wie in Abschnitt 7.2.1 diskutiert, zu vermeiden.

```
1  inline int inline_f ( int x )
2  { return x*x; }
3
4  int f( int x )
5  { return x*x; }
6
7  int main()
8  {
9    int y, x = 5;
10   y = f(x);
11   y = inline_f (x);
12   return 0;
13 }
```

Abbildung 7.12: Inline Beispiel

dies in Operatorsyntax als `&a + &b` schreibt. Das würde sehr unhandlich und verwirrend sein. Unter Benutzung von Referenzen jedoch, kann man es als `a + b` schreiben, was sehr natürlich aussieht.

7.2.3 Inline Funktionen

Inline Funktionen sind noch ein weiteres Merkmal von C++.[8] Inline Funktionen sind dazu bestimmt, die fehleranfälligen Präprozessor-basierten Makros, die Parameter erfordern, zu ersetzen. Erinnern wir uns, dass ein Makro in C, das eine Zahl quadriert, so aussehen könnte:

```
#define SQR(x) ((x)*(x))
```

Weil der Präprozessor kein C versteht und einfach ersetzt, sind die Klammern erforderlich, um in den meisten Fällen die korrekte Antwort zu berechnen. Jedoch selbst diese Version wird nicht die korrekte Antwort für `SQR(x++)` liefern.

Makros werden benutzt, weil sie den Overhead, für eine einfache Funktion einen Funktionsaufruf durchzuführen, eliminieren. Wie das Kapitel über Unterprogramme demonstrierte, erfordert die Durchführung eines Funktionsaufrufs mehrere Schritte. Für eine sehr einfache Funktion kann die Zeit, die es braucht, um den Funktionsaufruf zu machen, größer sein, als die Zeit, um die Operation in der Funktion tatsächlich auszuführen! Inline Funktionen sind ein viel freundlicherer Weg, um Code zu schreiben, der wie eine normale Funktion aussieht, aber *keinen* `CALL` eines gemeinsamen Codeblocks ausführt. Stattdessen werden Aufrufe von inline Funktionen durch Code ersetzt, der die Funktion ausführt. C++ erlaubt es, eine Funktion inline zu machen, indem das Schlüsselwort `inline` vor die Funktionsdefinition gesetzt wird. Betrachten wir zum Beispiel die in Abbildung 7.12 deklarierten Funktionen. Der Aufruf der Funktion `f` in Zeile 10 macht einen normalen Funktionsaufruf (in Assembler, unter der Annahme, dass `x` an Adresse `ebp−8` ist und `y` an `ebp−4` ist):

[8] C Compiler unterstützen oft dieses Merkmal als eine Erweiterung zu ANSI C.

```
1       push   dword [ebp-8]
2       call   _f
3       pop    ecx
4       mov    [ebp-4], eax
```

Jedoch würde der Aufruf der Funktion `inline_f` in Zeile 11 aussehen wie:

```
5       mov    eax, [ebp-8]
6       imul   eax, eax
7       mov    [ebp-4], eax
```

In diesem Fall gibt es zwei Vorteile fürs Inlining. Zuerst ist die Inlinefunktion schneller. Keine Parameter werden auf den Stack geschoben, kein Stackframe wird erzeugt und dann zerstört, kein Sprung wird ausgeführt. Zweitens benutzt die inline Funktion weniger Code! Der letzte Punkt trifft für dieses Beispiel zu, ist aber nicht in jedem Fall wahr.

Der Hauptnachteil von Inlining ist, dass inline Code nicht gelinkt wird, deshalb muss der Code einer Inlinefunktion für *alle* Dateien, die ihn benutzen, verfügbar sein. Das vorstehende Assemblercodebeispiel zeigt dies. Der Aufruf einer nicht-inline Funktion erfordert nur die Kenntnis der Parameter, des Typs des Rückgabewertes, Aufrufkonvention und den Namen des Labels für die Funktion. All diese Informationen sind im Prototypen der Funktion enthalten. Jedoch erfordert die Verwendung der Inlinefunktion Kenntnis vom gesamten Code der Funktion. Das bedeutet, dass wenn *irgendein* Teil der Inlinefunktion geändert wird, *alle* Quelldateien, die die Funktion benutzen, neu kompiliert werden müssen. Zur Erinnerung, wenn sich der Prototyp für nicht-inline Funktionen nicht ändert, müssen die Dateien, die die Funktion verwenden, oft nicht neu kompiliert werden. Aus all diesen Gründen wird der Code für Inlinefunktionen gewöhnlich in Headerdateien abgelegt. Diese Praxis steht im Gegensatz zu der normalerweise strengen und starren Regel in C, dass ausführbarer Code *niemals* in Headerdateien abgelegt wird.

7.2.4 Klassen

Eine C++ Klasse beschreibt den Typ eines *Objekts*. Ein Objekt hat sowohl Daten- als auch Funktionsmitglieder.[9] In anderen Worten, sie ist ein `struct` mit Daten und damit assoziierten Funktionen. Betrachten wir die einfache in Abbildung 7.13 definierte Klasse. Eine Variable vom Typ `Simple` würde genau wie ein normales C `struct` mit einem einzelnen `int`-Mitglied aussehen. Die Funktionen werden *nicht* im mit der Struktur assoziierten Speicher abgelegt. Jedoch unterscheiden sich Mitgliedsfunktionen von anderen Funktionen. Ihnen wird ein *verborgener* Parameter mitgegeben. Dieser Parameter ist ein Zeiger auf das Objekt, auf das die Mitgliedsfunktion einwirkt.

Tatsächlich benutzt C++ das `this` *Schlüsselwort, um innerhalb der Mitgliedsfunktion auf den Zeiger auf das zu bearbeitende Objekt zuzugreifen.*

Betrachten wir zum Beispiel die `set_data` Methode der `Simple`-Klasse von Abbildung 7.13. Wäre sie in C geschrieben, würde sie aussehen wie eine Funktion, der explizit ein Zeiger auf das Objekt, auf das sie einwirkt, mitgegeben

[9] In C++ oft *Mitgliedsfunktionen* genannt oder allgemeiner *Methoden*.

```
1  class Simple {
2  public:
3    Simple();                  // default Konstruktor
4    ~Simple();                 // Destruktor
5    int get_data() const;      // Mitglieds-Funktionen
6    void set_data( int );
7  private:
8    int data;                  // Daten-Mitglied
9  };
10
11 Simple::Simple()
12 { data = 0; }
13
14 Simple::~Simple()
15 { /* leerer Rumpf */ }
16
17 int Simple::get_data() const
18 { return data; }
19
20 void Simple::set_data( int x )
21 { data = x; }
```

Abbildung 7.13: Eine einfache C++ Klasse

```
1  void set_data( Simple *object, int x )
2  {
3    object->data = x;
4  }
```

Abbildung 7.14: C Version von Simple::set_data()

wurde, wie der Code in Abbildung 7.14 zeigt. Der `-S` Schalter des *DJGPP* Compilers (und genauso den *gcc* und Borland Compilern) veranlasst den Compiler ein Assemblerlisting zu generieren das das Assemblersprachenäquivalent des produzierten Codes enthält. Für *DJGPP* und *gcc* endet die Assemblerdatei in einer `.s` Erweiterung und unglücklicherweise benutzen AT&T eine Assemblersyntax, die ziemlich verschieden von den NASM und MASM Syntaxen[10] ist. (Borland und MS Compiler generieren eine Datei mit einer `.asm` Erweiterung unter Benutzung der MASM Syntax.) Abbildung 7.15 zeigt die Ausgabe von *DJGPP* in die NASM Syntax umgewandelt und mit Kommentaren versehen, die den Zweck der Anweisungen klarstellen. Beachte, dass in der allerersten Zeile der `set_data` Methode ein modifiziertes Label zugeordnet wird, das den Namen der Methode kodiert, den Namen der Klasse und die Parameter. Der Name der

[10] Der *gcc* Compiler beinhaltet seinen eigenen Assembler, *gas* genannt. Der *gas* Assembler verwendet AT&T Syntax und daher gibt der Compiler den Code im Format für *gas* aus. Es gibt mehrere Seiten im Web, die die Unterschiede in INTEL und AT&T Formaten diskutieren. Es gibt ebenso ein freies Programm namens `a2i` (http://www.multimania.com/placr/a2i.html), das AT&T Format ins NASM Format umwandelt.

```
1  _set_data__6Simplei:          ; dekorierter Name
2      push   ebp
3      mov    ebp, esp
4
5      mov    eax, [ebp+8]       ; eax = Zeiger aufs Objekt (this)
6      mov    edx, [ebp+12]      ; edx = Integer Parameter
7      mov    [eax], edx         ; Daten sind an Offset 0
8
9      leave
10     ret
```

Abbildung 7.15: Compiler-Ausgabe von Simple::set_data(int)

Klasse wird kodiert, weil andere Klassen eine Methode mit Namen `set_data` haben könnten und diese beiden Methoden *müssen* verschiedene Labels zugeordnet bekommen. Die Parameter sind kodiert, sodass die Klasse die `set_data` Methode überladen kann, um andere Parameter zu haben, genau wie normale C++ Funktionen. Jedoch, genau wie zuvor, werden verschiedene Compiler diese Informationen verschieden in den dekorierten Labels kodieren.

Als nächstes erscheint in Zeile 2 und 3 der bekannte Funktions-Prolog. In Zeile 5, wird der erste Parameter auf dem Stack nach `EAX` gespeichert. Dies ist *nicht* der `x` Parameter! Anstatt ihm, ist es der verborgene Parameter[11], der auf das Objekt zeigt, das bearbeitet wird. Zeile 6 speichert den `x` Parameter in `EDX` und Zeile 7 speichert `EDX` in das Doppelwort, auf das `EAX` zeigt. Das ist das `data` Mitglied des `Simple` Objekts, das, da es das einzige Datum in der Klasse ist, bei Offset 0 in der `Simple` Struktur gespeichert wird.

Beispiel

Dieser Abschnitt verwendet die Ideen des Kapitels, um eine C++ Klasse zu schaffen, die einen vorzeichenlosen Integer von beliebiger Größe repräsentiert. Da der Integer von jeder Größe sein kann, wird er in einem Array vorzeichenloser Integer (Doppelwörter) gespeichert. Ihm kann unter Benutzung dynamischer Speicherzuweisung jede Größe gegeben werden. Die Doppelwörter werden in umgekehrter Anordnung[12] gespeichert (d. h. das niederwertigste Doppelwort hat Index 0). Abbildung 7.16 zeigt die Definition der `Big_int` Klasse.[13] Die Größe eines `Big_int` wird durch die Größe des `unsigned` Arrays gemessen, der benutzt wird, um seine Daten zu speichern. Dem `size_` Daten-Mitglied der Klasse ist der Offset Null zugeordnet, und dem `number_` Mitglied ist Offset 4 zugeordnet.

Um diese Beispiele zu vereinfachen, können nur Objektinstanzen mit gleich großen Arrays zueinander addiert oder voneinander subtrahiert werden.

Die Klasse hat drei Konstruktoren: der erste (Zeile 9) initialisiert die Klasseninstanz unter Verwendung eines normalen vorzeichenlosen Integers; der zweite

[11] Wie üblich, ist im Assemblercode *nichts* verborgen!

[12] Warum? Weil Additionsoperationen dann immer am Anfang des Arrays beginnen und vorwärts schreiten.

[13] Siehe die Quellen der Code-Beispiele für den vollständigen Code dieses Beispiels. Der Text bezieht sich nur auf einen Teil des Codes.

```
1  class Big_int {
2  public:
3    /*
4     * Parameter:
5     *    size             – Größe des Integers als Anzahl von
6     *                       normalen unsigned ints ausgedrückt
7     *    initial_value    – Anfangswert von Big_int als normaler unsigned int
8     */
9    explicit Big_int( size_t      size,
10                     unsigned initial_value = 0 );
11   /*
12    * Parameter:
13    *    size             – Größe des Integers als Anzahl von
14    *                       normalen unsigned ints ausgedrückt
15    *    initial_value    – anfänglicher Wert von Big_int als ein String mit
16    *                       der hexadezimalen Repräsentation des Wertes.
17    */
18   Big_int( size_t          size,
19            const char * initial_value   );
20
21   Big_int( const Big_int &big_int_to_copy );
22   ~Big_int();
23
24   // gib Größe von Big_int zurück (in Einheiten von unsigned int's)
25   size_t size() const;
26
27   const Big_int &operator = ( const Big_int &big_int_to_copy );
28   friend Big_int operator + ( const Big_int &op1,
29                               const Big_int &op2 );
30   friend Big_int operator - ( const Big_int &op1,
31                               const Big_int &op2 );
32   friend bool operator == ( const Big_int &op1,
33                             const Big_int &op2 );
34   friend bool operator < ( const Big_int &op1,
35                            const Big_int &op2 );
36   friend ostream &operator << ( ostream      &os,
37                                 const Big_int &op );
38 private:
39   size_t        size_;     // Größe des unsigned Array
40   unsigned   *number_; // Zeiger auf unsigned Array mit dem Wert
41 };
```

Abbildung 7.16: Definition der `Big_int` Klasse

```
1  // Prototypen für Assembler Routinen
2  extern "C" {
3    int  add_big_ints ( Big_int        &res,
4                        const Big_int  &op1,
5                        const Big_int  &op2);
6    int  sub_big_ints ( Big_int        &res,
7                        const Big_int  &op1,
8                        const Big_int  &op2);
9  }
10
11 inline  Big_int  operator + ( const Big_int &op1, const Big_int &op2)
12 {
13   Big_int  result (op1.size ());
14   int  res = add_big_ints( result , op1, op2);
15   if (res == 1)
16     throw Big_int :: Overflow();
17   if (res == 2)
18     throw Big_int :: Size_mismatch();
19   return result ;
20 }
21
22 inline  Big_int  operator - ( const Big_int &op1, const Big_int &op2)
23 {
24   Big_int  result (op1.size ());
25   int  res = sub_big_ints( result , op1, op2);
26   if (res == 1)
27     throw Big_int :: Overflow();
28   if (res == 2)
29    throw Big_int :: Size_mismatch();
30   return result ;
31 }
```

Abbildung 7.17: Arithmetik Code der `Big_int` Klasse

(Zeile 18) initialisiert die Instanz unter Verwendung eines Strings, der einen hexadezimalen Wert enthält. Der dritte Konstruktor (Zeile 21) ist der *Kopierkonstruktor*.

Diese Diskussion konzentriert sich darauf, wie die Additions- und Subtraktionsoperatoren arbeiten, da diese es sind, wofür Assemblersprache verwendet wird. Abbildung 7.17 zeigt die relevanten Teile der Headerdateien dieser Operatoren. Sie zeigen, wie die Operatoren vorbereitet werden, um die Assemblerroutine aufzurufen. Da verschiedene Compiler radikal verschiedene Dekorationsregeln für Operatorfunktionen verwenden, werden inline Operatorfunktionen verwendet, um Calls zu Assemblerroutinen in C Bindung aufzusetzen. Das macht es relativ einfach, auf verschiedene Compiler zu portieren und ist fast so schnell wie direkte Aufrufe. Diese Technik eliminiert ebenso die Notwendigkeit von Assembler aus eine Exception auszulösen!

Warum wird hier überhaupt Assembler verwendet? Erinnern wir uns, dass, um Multipräzisions-Arithmetik durchzuführen, der Übertrag von einem Doppelwort zum nächst signifikanten Doppelwort addiert werden muss. C++ (und C) erlauben dem Programmierer nicht, auf das Carryflag der CPU zuzugreifen. Die Durchführung der Addition könnte nur getan werden, indem C++ unabhängig das Carryflag entwickelt und es bedingt zum nächsten Doppelwort addiert. Es ist sehr viel effizienter, den Code in Assembler zu schreiben, wo auf das Carryflag zugegriffen werden kann, und die Benutzung des `ADC` Befehls, der automatisch das Carryflag dazuaddiert, macht viel Sinn.

Zur Kürze wird hier nur die `add_big_ints` Assemblerroutine besprochen. Unten ist der Code für diese Routine (aus `big_math.asm`):

big_math.asm

```
1  segment .text
2  global  add_big_ints, sub_big_ints
3  %define size_offset 0
4  %define number_offset 4
5
6  %define EXIT_OK 0
7  %define EXIT_OVERFLOW 1
8  %define EXIT_SIZE_MISMATCH 2
9
10 ; Parameter für add und sub Routinen
11 %define res ebp+8
12 %define op1 ebp+12
13 %define op2 ebp+16
14
15 add_big_ints:
16     push    ebp
17     mov     ebp, esp
18     push    ebx
19     push    esi
20     push    edi
21     ;
22     ; zuerst setze esi um auf op1 zu zeigen
23     ;              edi um auf op2 zu zeigen
24     ;              ebx um auf res zu zeigen
```

```
25      mov     esi, [op1]
26      mov     edi, [op2]
27      mov     ebx, [res]
28      ;
29      ; stelle sicher, dass alle 3 Big_int's die gleiche Größe haben
30      ;
31      mov     eax, [esi+size_offset]
32      cmp     eax, [edi+size_offset]
33      jne     sizes_not_equal          ; op1.size_ != op2.size_
34      cmp     eax, [ebx+size_offset]
35      jne     sizes_not_equal          ; op1.size_ != res.size_
36
37      mov     ecx, eax                 ; ecx = Größe der Big_int's
38      ;
39      ; nun setze Register, damit sie auf ihre entsprechenden Arrays zeigen
40      ;       esi = op1.number_
41      ;       edi = op2.number_
42      ;       ebx = res.number_
43      ;
44      mov     ebx, [ebx+number_offset]
45      mov     esi, [esi+number_offset]
46      mov     edi, [edi+number_offset]
47
48      clc                              ; lösche Carry Flag
49      xor     edx, edx                 ; edx = 0
50      ;
51      ; Additions Schleife
52  add_loop:
53      mov     eax, [edi+4*edx]
54      adc     eax, [esi+4*edx]
55      mov     [ebx+4*edx], eax
56      inc     edx                      ; ändert Carry Flag nicht
57      loop    add_loop
58
59      jc      overflow
60  ok_done:
61      xor     eax, eax                 ; Rückgabewert = EXIT_OK
62      jmp     done
63  overflow:
64      mov     eax, EXIT_OVERFLOW
65      jmp     done
66  sizes_not_equal:
67      mov     eax, EXIT_SIZE_MISMATCH
68  done:
69      pop     edi
70      pop     esi
71      pop     ebx
72      leave
73      ret
```

big_math.asm

```
1  #include " big_int .hpp"
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7    try {
8      Big_int b(5, "8000000000000a00b");
9      Big_int a(5, "80000000000010230");
10     Big_int c = a + b;
11     cout << a << " + " << b << " = " << c << endl;
12     for( int i=0; i < 2; i++ ) {
13       c = c + a;
14       cout << "c = " << c << endl;
15     }
16     cout << "c-1 = " << c - Big_int(5, 1) << endl;
17     Big_int d(5, "12345678");
18     cout << "d = " << d << endl;
19     cout << "c == d " << (c == d) << endl;
20     cout << "c > d " << (c > d) << endl;
21   }
22   catch( const char * str ) {
23     cerr << "Caught: " << str << endl;
24   }
25   catch( Big_int :: Overflow ) {
26     cerr << "Overflow" << endl;
27   }
28   catch( Big_int :: Size_mismatch ) {
29     cerr << "Size mismatch" << endl;
30   }
31   return 0;
32 }
```

Abbildung 7.18: Einfache Anwendung von `Big_int`

Es wird gehofft, dass das meiste diesen Codes für den Leser inzwischen einfach sein sollte. Zeilen 16 bis 27 speichern Zeiger zu den der Funktion übergebenen `Big_int` Objekten in Registern. Denken wir daran, dass Referenzen wirklich nur Zeiger sind. Zeilen 31 bis 35 testen, um sicher zu stellen, dass die Größen der Arrays der drei Objekte gleich sind. (Beachte, dass der Offset von `size_` zum Zeiger dazugezählt wird, um auf das Daten-Mitglied zuzugreifen.) Zeilen 44 bis 46 passen die Register an, um auf den durch das entsprechende Objekt verwendeten Array zu zeigen, anstatt auf die Objekte selbst. (Wieder wird der Offset des `number_` Members zum Zeiger auf das Objekt dazugezählt.)

Die Schleife in Zeilen 52 bis 57 addiert die im Array gespeicherten Integer zueinander, indem die niederwertigsten Doppelwörter zuerst, dann die nächst niederwertigen Doppelwörter, usw. addiert werden. Die Addition muss für Multipräzisions-Arithmetik in dieser Reihenfolge durchgeführt werden (siehe Ab-

schnitt 2.1.5). Zeile 59 testet auf Überlauf. Bei Überlauf wird das Carryflag durch die letzte Addition des höchstwertigen Doppelworts gesetzt sein. Da die Doppelwörter im Array in little endian Ordnung gespeichert sind, beginnt die Schleife am Anfang des Arrays und wandert vorwärts dem Ende zu.

Abbildung 7.18 zeigt ein kurzes Beispiel, das die Klasse `Big_int` verwendet. Beachte, dass `Big_int` Konstanten explizit wie in Zeile 16 deklariert werden müssen. Das ist aus zwei Gründen notwendig. Zuerst gibt es keinen Konversionskonstruktor, der einen vorzeichenlosen `int` zu einem `Big_int` konvertieren wird. Zweitens können nur `Big_int`s der gleichen Größe zusammengezählt werden. Das macht Konversionen problematisch, da es schwierig sein würde, zu wissen, zu welcher Größe zu konvertieren sei. Eine anspruchsvollere Implementierung der Klasse würde es erlauben jede Größe zu jeder anderen Größe zu addieren. Der Autor wollte dieses Beispiel nicht zu sehr komplizieren dadurch, dass er dies hier implementierte. (Jedoch wird der Leser ermutigt, dies zu tun.)

7.2.5 Vererbung und Polymorphismus

Vererbung erlaubt einer Klasse, die Daten und Methoden einer anderen Klasse zu erben. Betrachten wir zum Beispiel den Code in Abbildung 7.19. Er zeigt zwei Klassen, `A` und `B`, wobei Klasse `B` von `A` erbt. Die Ausgabe des Programms ist:

```
Size of a: 4 Offset of ad: 0
Size of b: 8 Offset of ad: 0 Offset of bd: 4
A::m()
A::m()
```

Beachte, dass die `ad` Daten-Mitglieder beider Klassen (`B` erbt sie von `A`) den gleichen Offset haben. Das ist wichtig, da der Funktion `f` ein Zeiger übergeben werden könnte auf entweder ein `A` Objekt oder jedes Objekt mit einem von `A` abgeleiteten (d. h. geerbten) Typ. Abbildung 7.20 zeigt den (editierten) Assemblercode für die Funktion (von *gcc* erzeugt).

Der Ausgabe können wir entnehmen, dass die Methode `m` von `A` für beide Objekte, `a` und `b`, aufgerufen wurde. Im Assemblercode kann man sehen, dass der Aufruf von `A::m()` hart in die Funktion kodiert ist. Für wahre objektorientierte Programmierung sollte die aufgerufene Methode davon abhängen, welcher Objekttyp an die Funktion übergeben wird. Dies ist als *Polymorphismus* bekannt. C++ schaltet dieses Merkmal standardmäßig ab. Man benutzt das Schlüsselwort *virtual*, um es verwenden zu können. Abbildung 7.21 zeigt, wie die beiden Klassen geändert werden würden. Von dem anderen Code muss nichts geändert werden. Polymorphismus kann auf viele Weisen implementiert werden. Unglücklicherweise ist *gcc*s Implementierung zum Zeitpunkt dieses Schreibens im Fluss und wird signifikant komplizierter als seine ursprüngliche Implementierung werden. Im Interesse, diese Diskussion zu vereinfachen, möchte der Autor nur die Implementierungen des Polymorphismus abdecken, welche die Windows basierten Compiler von Microsoft und Borland verwenden. Diese Implementierung hat sich in vielen Jahren nicht geändert und wird sich wahrscheinlich in der vorhersehbaren Zukunft auch nicht ändern.

```
1  #include <cstddef>
2  #include <iostream>
3  using namespace std;
4
5  class A {
6  public:
7    void __cdecl m() { cout << "A::m()" << endl; }
8    int ad;
9  };
10
11 class B : public A {
12 public:
13   void __cdecl m() { cout << "B::m()" << endl; }
14   int bd;
15 };
16
17 void f( A *p )
18 {
19   p->ad = 5;
20   p->m();
21 }
22
23 int main()
24 {
25   A a;
26   B b;
27   cout << "Size of a: " << sizeof(a)
28        << " Offset of ad: " << offsetof(A, ad) << endl;
29   cout << "Size of b: " << sizeof(b)
30        << " Offset of ad: " << offsetof(B, ad)
31        << " Offset of bd: " << offsetof(B, bd) << endl;
32   f(&a);
33   f(&b);
34   return 0;
35 }
```

Abbildung 7.19: Einfache Vererbung

```
1  _f__FP1A:                    ; dekorierter Funktionsname
2       push   ebp
3       mov    ebp, esp
4       mov    eax, [ebp+8]      ; eax zeigt auf Objekt
5       mov    dword [eax], 5    ; benutze Offset 0 für ad
6       mov    eax, [ebp+8]      ; übergebe Adresse von Objekt an A::m()
7       push   eax
8       call   _m__1A            ; dekorierter Methodenname für A::m()
9       add    esp, 4
10      leave
11      ret
```

Abbildung 7.20: Assemblercode für einfache Vererbung

```
1  class A {
2  public:
3    virtual void __cdecl m() { cout << "A::m()" << endl; }
4    int ad;
5  };
6
7  class B : public A {
8  public:
9    virtual void __cdecl m() { cout << "B::m()" << endl; }
10   int bd;
11 };
```

Abbildung 7.21: Polymorphe Vererbung

```
1  ?f@@YAXPAVA@@@Z:
2      push   ebp
3      mov    ebp, esp
4
5      mov    eax, [ebp+8]
6      mov    dword [eax+4], 5  ; p->ad = 5;
7
8      mov    ecx, [ebp+8]      ; ecx = p
9      mov    edx, [ecx]        ; edx = Zeiger auf vtable
10     mov    eax, [ebp+8]      ; eax = p
11     push   eax               ; push "this" Zeiger
12     call   dword [edx]       ; rufe erste Funktion in vtable auf
13     add    esp, 4            ; räume Stack auf
14
15     pop    ebp
16     ret
```

Abbildung 7.22: Assemblercode für Funktion `f()`

Mit diesen Änderungen ändert sich die Ausgabe des Programms:

```
Size of a: 8 Offset of ad: 4
Size of b: 12 Offset of ad: 4 Offset of bd: 8
A::m()
B::m()
```

Nun ruft der zweite Aufruf von `f` die Methode `B::m()` auf, weil ihr ein `B` Objekt übergeben wurde. Das ist jedoch nicht die einzige Änderung. Die Größe eines `A` ist jetzt 8 (und `B` ist 12). Ebenso ist der Offset von `ad` 4, nicht 0. Was ist an Offset 0? Die Antworten auf diese Fragen stehen in Beziehung damit, wie Polymorphismus implementiert ist.

Einer C++ Klasse, die irgendeine virtuelle Methode besitzt, wird ein zusätzliches verborgenes Feld gegeben, das ein Zeiger auf einen Array von Methodenzeigern[14] ist. Diese Tabelle wird oft die *vtable* genannt. Für die Klassen `A` und `B` wird dieser Zeiger bei Offset 0 gespeichert. Die Windows Compiler legen diesen Zeiger immer an den Anfang der Klasse an die Spitze des Vererbungsbaumes. Indem man sich den Assemblercode (Abbildung 7.22) ansieht, der für die Funktion `f` (aus Abbildung 7.19), die die virtuelle Methodenversion des Programms ist, generiert wurde, kann man sehen, dass der Aufruf von Methode `m` nicht zu einem Label ist. Zeile 9 findet die Adresse der Vtable des Objekts. Die Adresse des Objekts wird in Zeile 11 auf den Stack gelegt. Zeile 12 ruft die virtuelle Methode auf, indem sie zur ersten Adresse in der Vtable[15] verzweigt. Dieser Aufruf benutzt kein Label, er springt zu der Codeadresse, auf die `EDX` zeigt. Dieser Typ von Aufruf ist ein Beispiel einer *späten Bindung*. Späte Bindung verzögert die

[14] Für Klassen ohne virtuelle Methoden machen C++ Compiler die Klassen immer kompatibel zu einem normalen C struct mit den gleichen Daten-Mitgliedern.

[15] Natürlich ist dieser Wert bereits im `ECX` Register. Es kam dort in Zeile 8 hinein und Zeile 10 könnte entfernt und die nächste Zeile geändert werden um `ECX` auf den Stack zu schieben. Der Code ist nicht sehr effizient, weil er ohne Compileroptimierungen generiert wurde.

```
1  class A {
2  public:
3    virtual void __cdecl m1() { cout << "A::m1()" << endl; }
4    virtual void __cdecl m2() { cout << "A::m2()" << endl; }
5    int ad;
6  };
7
8  class B : public A {     // B erbt m2() von A
9  public:
10   virtual void __cdecl m1() { cout << "B::m1()" << endl; }
11   int bd;
12 };
13 /* gibt die vtable des gegebenen Objekts aus */
14 void print_vtable ( A *pa )
15 {
16   // p sieht pa als einen Array von Dwords
17   unsigned *p = reinterpret_cast<unsigned *>(pa);
18   // vt sieht vtable als einen Array von Zeigern
19   void ** vt = reinterpret_cast<void **>(p[0]);
20   cout << hex << "vtable address = " << vt << endl;
21   for( int i=0; i < 2; i++ )
22     cout << "dword " << i << ": " << vt[i] << endl;
23
24   // rufe virtuelle Funktionen in EXTREM nicht-portabler Weise auf!
25   void (*m1func_pointer)(A *);    // Funktionszeiger Variable
26   m1func_pointer = reinterpret_cast<void (*)(A*)>(vt[0]);
27   m1func_pointer(pa);             // call Methode m1 via Funktionszeiger
28
29   void (*m2func_pointer)(A *);    // Funktionszeiger Variable
30   m2func_pointer = reinterpret_cast<void (*)(A*)>(vt[1]);
31   m2func_pointer(pa);             // call Methode m2 via Funktionszeiger
32 }
33
34 int main()
35 {
36   A a;    B b1;  B b2;
37   cout << "a: " << endl;   print_vtable (&a);
38   cout << "b1: " << endl;  print_vtable (&b);
39   cout << "b2: " << endl;  print_vtable (&b2);
40   return 0;
41 }
```

Abbildung 7.23: Komplizierteres Beispiel

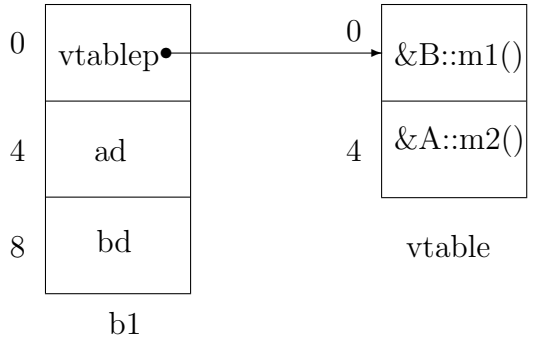


Abbildung 7.24: Interne Repräsentation von `b1`

```
a:
vtable address = 004120E8
dword 0: 00401320
dword 1: 00401350
A::m1()
A::m2()
b1:
vtable address = 004120F0
dword 0: 004013A0
dword 1: 00401350
B::m1()
A::m2()
b2:
vtable address = 004120F0
dword 0: 004013A0
dword 1: 00401350
B::m1()
A::m2()
```

Abbildung 7.25: Ausgabe des Programms in Abbildung 7.23

Entscheidung, welche Methode aufgerufen wird, bis der Code läuft. Das erlaubt dem Code, die passende Methode für das Objekt aufzurufen. Der Normalfall (Abbildung 7.20) kodiert einen Aufruf zu einer bestimmten Methode hart und wird *frühe Bindung* genannt (da hier die Methode früh, zur Kompilierzeit gebunden wird).

Der aufmerksame Leser wird sich wundern, warum die Klassenmethode in Abbildung 7.21 explizit deklariert wurde um die C Aufrufkonvention zu benutzen, indem das Schlüsselwort `__cdecl` benutzt wird. Standardmäßig verwendet Microsoft für C++ Klassenmethoden eine von der Standard C Konvention unterschiedliche Aufrufkonvention. Sie übergibt den Zeiger auf das durch die Methode zu bearbeitende Objekt im `ECX` Register anstatt den Stack zu benutzen. Der Stack wird immer noch für die anderen expliziten Parameter der Methode benutzt. Der `__cdecl` Modifizierer teilt ihm mit, die Standard C Aufrufkonvention zu benutzen. Borland C++ benutzt standardmäßig die C Aufrufkonvention.

Sehen wir uns als nächstes ein etwas komplizierteres Beispiel an (Abbildung 7.23). In ihm haben die Klassen `A` und `B` jeweils zwei Methoden: `m1` und `m2`. Denken wir daran, da die Klasse `B` keine eigene `m2` Methode definiert, sie die Methode von Klasse `A` erbt. Abbildung 7.24 zeigt, wie das `b1` Objekt im Speicher erscheint. Abbildung 7.25 zeigt die Ausgabe des Programms. Zuerst betrachten wir die Adresse der Vtable jeden Objekts. Die Adressen der beiden `B` Objekte sind dieselben und deshalb teilen sie sich dieselbe Vtable. Eine Vtable ist Eigentum einer Klasse, nicht eines Objekts (wie ein `static` Daten-Mitglied). Als nächstes sehen wir nach den Adressen in den Vtables. Bei Betrachtung der Assemblerausgabe kann man feststellen, dass der Methodenzeiger von `m1` an Offset 0 liegt (oder Doppelwort 0) und `m2` ist an Offset 4 (Doppelwort 1). Die `m2` Methodenzeiger sind dieselben für die Vtables der `A` und `B` Klassen, weil Klasse `B` die `m2` Methode von der Klasse `A` erbt.

Zeilen 25 bis 31 zeigen, wie man eine virtuelle Funktion aufrufen könnte, indem man ihre Adresse der Vtable für das Objekt[16] ausliest. Die Methodenadresse wird über einen expliziten *this* Zeiger in einem C-Typ Funktionszeiger gespeichert. Aus der Ausgabe in Abbildung 7.25 kann man sehen, dass es funktioniert. Schreiben Sie jedoch bitte *keinen* Code wie diesen! Das wurde nur verwendet, um zu illustrieren, wie die virtuellen Methoden die Vtable benutzen.

Es gibt einige praktische Lektionen daraus zu lernen. Eine wichtige Tatsache ist, dass man sehr vorsichtig sein muss, wenn man Klassenvariable in eine binäre Datei liest und schreibt. Man kann nicht gerade auf das gesamte Objekt ein binäres Read oder Write benutzen, da dies den Vtable-Zeiger von der Datei lesen oder hinein schreiben würde! Das ist ein Zeiger darauf, wo die Vtable im Speicher des Programms liegt und wird sich von Programm zu Programm ändern. Das gleiche Problem kann in C mit Structs auftreten, nur haben, in C, Structs nur dann Zeiger in sich, wenn der Programmierer sie explizit hineintut. Jedoch sind in keiner der Klassen `A` oder `B` offensichtliche Zeiger definiert.

Wiederum ist es wichtig, sich klar zu machen, dass verschiedene Compiler virtuelle Methoden verschieden implementieren. In Windows benutzen COM (Component Object Model) Klassenobjekte Vtables um COM Schnittstellen[17] zu implementieren. Nur Compiler, die Vtables für virtuelle Methoden implementieren, so wie Microsoft es tut, können COM Klassen erzeugen. Das ist der Grund, warum Borland die gleiche Implementierung verwendet wie Microsoft und einer der Gründe, warum *gcc* nicht verwendet werden kann um COM Klassen zu erzeugen.

Der Code für die virtuelle Methode sieht genauso aus wie der einer nichtvirtuellen. Nur der aufrufende Code ist unterschiedlich. Wenn der Compiler absolut sicher sein kann, welche virtuelle Methode aufgerufen wird, kann er die Vtable ignorieren und die Methode direkt aufrufen (d. h. benutzt frühe Bindung).

7.2.6 Andere C++ Merkmale

Die Arbeitsweisen anderer C++ Merkmale (z. B. RunTime Type Information, Ausnahmebehandlung und Mehrfachvererbung) gehen über den Rahmen dieses

[16] Zur Erinnerung, dieser Code funktioniert nur mit den Compilern von MS und Borland, nicht mit *gcc*.

[17] COM Klassen benutzen ebenfalls die `__stdcall` Aufrufkonvention, nicht die von Standard C.

Textes hinaus. Wenn der Leser weiter gehen möchte, ist ein guter Ausgangspunkt *The Annotated C++ Reference Manual* von Ellis und Stroustrup und *The Design and Evolution of C++* von Stroustrup.

Anhang A

80x86 Befehle

A.1 Nicht Fließpunkt-Befehle

Dieser Abschnitt listet und beschreibt die Wirkungen und Formate der nicht-Fließpunktbefehle der Intel 80x86 CPU Familie.

Die Formate benutzen die folgenden Abkürzungen:

R	Allzweck Register
R8	8-bit Register
R16	16-bit Register
R32	32-bit Register
SR	Selektor Register
M	Speicher
M8	Byte
M16	Wort
M32	Doppelwort
I	unmittelbarer Wert

Diese können für die Befehle mit mehreren Operanden kombiniert werden. Zum Beispiel meint das Format *R, R*, dass der Befehle zwei Registeroperanden verwendet. Viele der Befehle mit zwei Operanden erlauben den gleichen Operanden. Die Abkürzung *O2* wird benutzt, um diese Operanden zu repräsentieren: *R,R R,M R,I M,R M,I*. Wenn ein 8-bit Register oder Speicher für einen Operanden benutzt werden kann, wird die Abkürzung *R/M8* verwendet.

Die Tabelle zeigt auch, wie verschiedene Bits des FLAGS Registers durch jeden Befehl beeinflusst werden. Wenn die Spalte leer ist, wird das entsprechende Bit überhaupt nicht beeinflusst. Wenn das Bit immer zu einem bestimmten Wert geändert wird, wird eine 1 oder 0 in der Spalte angezeigt. Wenn das Bit zu einem geändert wird, das von den Operanden der Instruktion abhängt, wird ein *C* in die Spalte gesetzt. Schließlich, wenn das Bit in einer undefinierten Weise modifiziert wird, erscheint ein *?* in der Spalte. Weil die einzigen Befehle, die das Richtungsbit ändern, `CLD` und `STD` sind, wird es nicht unter den FLAGS Spalten aufgeführt.

Befehl	Beschreibung	Formate	Flags					
			O	S	Z	A	P	C
ADC	Add with Carry	O2	C	C	C	C	C	C
ADD	Add Integers	O2	C	C	C	C	C	C
AND	Bitwise AND	O2	0	C	C	?	C	0
BSWAP	Byte Swap	R32						
CALL	Call Routine	R M I						
CBW	Convert Byte to Word							
CDQ	Convert Dword EAX to Qword EDX:EAX							
CLC	Clear Carry							0
CLD	Clear Direction Flag							
CMC	Complement Carry							C
CMP	Compare Integers	O2	C	C	C	C	C	C
CMPSB	Compare Bytes		C	C	C	C	C	C
CMPSD	Compare Dwords		C	C	C	C	C	C
CMPSW	Compare Words		C	C	C	C	C	C
CWD	Convert Word AX to Dword DX:AX							
CWDE	Convert Word AX to Dword EAX							
DEC	Decrement Integer	R M	C	C	C	C	C	
DIV	Unsigned Divide	R M	?	?	?	?	?	?
ENTER	Make Stack Frame	I,0						
IDIV	Signed Divide	R M	?	?	?	?	?	?
IMUL	Signed Multiply	R M R16,R/M16 R32,R/M32 R16,I R32,I R16,R/M16,I R32,R/M32,I	C	?	?	?	?	C
INC	Increment Integer	R M	C	C	C	C	C	
INT	Generate Interrupt	I						
JA	Jump Above	I						
JAE	Jump Above or Equal	I						
JB	Jump Below	I						
JBE	Jump Below or Equal	I						
JC	Jump Carry	I						
JCXZ	Jump if CX = 0	I						
JE	Jump Equal	I						
JECXZ	Jump if ECX = 0	I						
JG	Jump Greater	I						
JGE	Jump Greater or Equal	I						
JL	Jump Less	I						
JLE	Jump Less or Equal	I						
JMP	Unconditional Jump	R M I						
JNA	Jump Not Above	I						
JNAE	Jump Not Above or Equal	I						

Befehl	Beschreibung	Formate	Flags					
			O	S	Z	A	P	C
JNB	Jump Not Below	I						
JNBE	Jump Not Below or Equal	I						
JNC	Jump No Carry	I						
JNE	Jump Not Equal	I						
JNG	Jump Not Greater	I						
JNGE	Jump Not Greater or Equal	I						
JNL	Jump Not Less	I						
JNLE	Jump Not Less or Equal	I						
JNO	Jump No Overflow	I						
JNS	Jump No Sign	I						
JNZ	Jump Not Zero	I						
JO	Jump Overflow	I						
JPE	Jump Parity Even	I						
JPO	Jump Parity Odd	I						
JS	Jump Sign	I						
JZ	Jump Zero	I						
LAHF	Load FLAGS into AH							
LEA	Load Effective Address	R32,M						
LEAVE	Release Stack Frame							
LODSB	Load Byte							
LODSD	Load Dword							
LODSW	Load Word							
LOOP	Loop	I						
LOOPE/LOOPZ	Loop If Equal	I						
LOOPNE/LOOPNZ	Loop If Not Equal	I						
MOV	Move Data	O2 SR,R/M16 R/M16,SR						
MOVSB	Move Byte							
MOVSD	Move Dword							
MOVSW	Move Word							
MOVSX	Move Signed	R16,R/M8 R32,R/M8 R32,R/M16						
MOVZX	Move Unsigned	R16,R/M8 R32,R/M8 R32,R/M16						
MUL	Unsigned Multiply	R M	C	?	?	?	?	C
NEG	Negate	R M	C	C	C	C	C	C
NOP	No Operation							
NOT	1's Complement	R M						
OR	Bitwise OR	O2	0	C	C	?	C	0
POP	Pop from Stack	R/M16 R/M32						
POPA	Pop All							

Befehl	Beschreibung	Formate	Flags					
			O	S	Z	A	P	C
POPF	Pop FLAGS		C	C	C	C	C	C
PUSH	Push to Stack	R/M16						
		R/M32 I						
PUSHA	Push All							
PUSHF	Push FLAGS							
RCL	Rotate Left with Carry	R/M,I	C					C
		R/M,CL						
RCR	Rotate Right with Carry	R/M,I	C					C
		R/M,CL						
REP	Repeat							
REPE/REPZ	Repeat If Equal							
REPNE/REPNZ	Repeat If Not Equal							
RET	Return							
ROL	Rotate Left	R/M,I	C					C
		R/M,CL						
ROR	Rotate Right	R/M,I	C					C
		R/M,CL						
SAHF	Copy AH into FLAGS			C	C	C	C	C
SAL	Shift to Left	R/M,I						C
		R/M,CL						
SAR	Arithmetic Shift to Right	R/M,I						C
		R/M,CL						
SBB	Subtract with Borrow	O2	C	C	C	C	C	C
SCASB	Scan for Byte		C	C	C	C	C	C
SCASD	Scan for Dword		C	C	C	C	C	C
SCASW	Scan for Word		C	C	C	C	C	C
SETA	Set Above	R/M8						
SETAE	Set Above or Equal	R/M8						
SETB	Set Below	R/M8						
SETBE	Set Below or Equal	R/M8						
SETC	Set Carry	R/M8						
SETE	Set Equal	R/M8						
SETG	Set Greater	R/M8						
SETGE	Set Greater or Equal	R/M8						
SETL	Set Less	R/M8						
SETLE	Set Less or Equal	R/M8						
SETNA	Set Not Above	R/M8						
SETNAE	Set Not Above or Equal	R/M8						
SETNB	Set Not Below	R/M8						
SETNBE	Set Not Below or Equal	R/M8						
SETNC	Set No Carry	R/M8						
SETNE	Set Not Equal	R/M8						
SETNG	Set Not Greater	R/M8						
SETNGE	Set Not Greater or Equal	R/M8						
SETNL	Set Not Less	R/M8						
SETNLE	Set Not Less or Equal	R/M8						
SETNO	Set No Overflow	R/M8						

Befehl	Beschreibung	Formate	Flags					
			O	**S**	**Z**	**A**	**P**	**C**
SETNS	Set No Sign	R/M8						
SETNZ	Set Not Zero	R/M8						
SETO	Set Overflow	R/M8						
SETPE	Set Parity Even	R/M8						
SETPO	Set Parity Odd	R/M8						
SETS	Set Sign	R/M8						
SETZ	Set Zero	R/M8						
SHR	Logical Shift to Right	R/M,I						C
		R/M,CL						
SHL	Logical Shift to Left	R/M,I						C
		R/M,CL						
STC	Set Carry							1
STD	Set Direction Flag							
STOSB	Store Byte							
STOSD	Store Dword							
STOSW	Store Word							
SUB	Subtract	O2	C	C	C	C	C	C
TEST	Logical Compare	R/M,R	0	C	C	?	C	0
		R/M,I						
XCHG	Exchange	R/M,R						
		R,R/M						
XOR	Bitwise XOR	O2	0	C	C	?	C	0

A.2 Fließpunkt-Befehle

In diesem Abschnitt werden viele der Befehle des 80x87 mathematischen Coprozessors beschrieben. Der Beschreibungsabschnitt beschreibt kurz die Operation des Befehls. Um Platz zu sparen, wird die Information, ob der Befehl den Wert vom TOS entfernt, in der Beschreibung nicht angegeben.

Die Formatspalte zeigt, welcher Typ von Operand mit jedem Befehl benutzt werden kann. Die folgenden Abkürzungen werden verwendet:

STn	ein Coprozessor Register
F	einfach genaue Zahl im Speicher
D	doppelt genaue Zahl im Speicher
E	extended genaue Zahl im Speicher
I16	Integer Wort im Speicher
I32	Integer Doppelwort im Speicher
I64	Integer Quadwort im Speicher

Befehle, die einen Pentium Pro oder besser erfordern, sind mit einem Asteriskus (*) markiert.

Befehl	Beschreibung	Formate
FABS	ST0 = |ST0|	
FADD *src*	ST0 += *src*	STn F D
FADD *dest*, ST0	*dest* += ST0	STn
FADDP *dest*[,ST0]	*dest* += ST0	STn
FCHS	ST0 = −ST0	
FCOM *src*	Compare ST0 and *src*	STn F D
FCOMI* *src*	Compare into FLAGS	STn
FCOMIP* *src*	Compare into FLAGS	STn
FCOMP *src*	Compare ST0 and *src*	STn F D
FCOMPP *src*	Compare ST0 and ST1	
FDIV *src*	ST0 /= *src*	STn F D
FDIV *dest*, ST0	*dest* /= ST0	STn
FDIVP *dest*[,ST0]	*dest* /= ST0	STn
FDIVR *src*	ST0 = *src*/ST0	STn F D
FDIVR *dest*, ST0	*dest* = ST0/*dest*	STn
FDIVRP *dest*[,ST0]	*dest* = ST0/*dest*	STn
FFREE *dest*	Mark as Empty	STn
FIADD *src*	ST0 += *src*	I16 I32
FICOM *src*	Compare ST0 and *src*	I16 I32
FICOMP *src*	Compare ST0 and *src*	I16 I32
FIDIV *src*	ST0 /= *src*	I16 I32
FIDIVR *src*	ST0 = *src*/ST0	I16 I32
FILD *src*	Push *src* on Stack	I16 I32 I64
FIMUL *src*	ST0 *= *src*	I16 I32
FINIT	Initialize Coprocessor	
FIST *dest*	Store ST0	I16 I32
FISTP *dest*	Store ST0	I16 I32 I64
FISUB *src*	ST0 -= *src*	I16 I32
FISUBR *src*	ST0 = *src* - ST0	I16 I32

Befehl	Beschreibung	Formate
`FLD` *src*	Push *src* on Stack	STn F D E
`FLD1`	Push 1.0 on Stack	
`FLDCW` *src*	Load Control Word Register	I16
`FLDPI`	Push π on Stack	
`FLDZ`	Push 0.0 on Stack	
`FMUL` *src*	`ST0 *=` *src*	STn F D
`FMUL` *dest*`, ST0`	*dest* `*= ST0`	STn
`FMULP` *dest*`[,ST0]`	*dest* `*= ST0`	STn
`FRNDINT`	Round `ST0`	
`FSCALE`	`ST0` $= \texttt{ST0} \times 2^{\lfloor \texttt{ST1} \rfloor}$	
`FSQRT`	`ST0` $= \sqrt{\texttt{ST0}}$	
`FST` *dest*	Store `ST0`	STn F D
`FSTCW` *dest*	Store Control Word Register	I16
`FSTP` *dest*	Store `ST0`	STn F D E
`FSTSW` *dest*	Store Status Word Register	I16 AX
`FSUB` *src*	`ST0 -=` *src*	STn F D
`FSUB` *dest*`, ST0`	*dest* `-= ST0`	STn
`FSUBP` *dest*`[,ST0]`	*dest* `-= ST0`	STn
`FSUBR` *src*	`ST0 =` *src* `- ST0`	STn F D
`FSUBR` *dest*`, ST0`	*dest* `= ST0 -` *dest*	STn
`FSUBRP` *dest*`[,ST0]`	*dest* `= ST0 -` *dest*	STn
`FTST`	Compare `ST0` with 0.0	
`FXCH` *dest*	Exchange `ST0` and *dest*	STn

Index