Assembler-Programmierung für x86-Prozessoren

de.wikibooks.org

31. August 2014

Inhaltsverzeichnis

1 Einleitung

1.1 Wofür eignet sich Assembler?

Maschinensprache ist die Sprache, die der Prozessor versteht. Ein Programm in Maschinensprache besteht im Grunde nur aus Dualzahlen.

Als Beispielprogramm sehen Sie die ersten 8 Byte des 1. Sektors jeder Festplatte, den Beginn des so genannten „Urladeprogramms“. In der ersten Spalte steht die Speicheradresse in hexadezimaler Schreibweise, in der zweiten Spalte der Inhalt des Speicherplatzes, und zwar in binärer Darstellung:

```
0000   11111010
0001   00110011
0002   11000000
0003   10001110
0004   11010000
0005   10111100
0006   00000000
0007   01111100
```

Diese Programmdarstellung ist sehr unpraktisch. Die Kolonnen von Einsen und Nullen sind unübersichtlich, sehr aufwändig zu schreiben, sind fehleranfällig bei der Eingabe und nehmen unverhältnismäßig viel Platz auf dem Papier ein. Deshalb ist es üblich, die gleichen Daten in hexadezimaler Darstellung aufzulisten. Sie sehen hier vom gleichen Programm den Anfang, und zwar nicht nur acht Byte, sondern die ersten 32 Byte:

```
0000   FA 33 C0 8E D0 BC 00 7C 8B F4 50 07 50 1F FB FC
0010   BF 00 06 B9 00 01 F2 A5 EA 1D 06 00 00 BE BE 07
```

Weil die hexadezimale Schreibweise kompakter ist, hat es sich eingebürgert, wie in diesem Beispiel in jede Zeile den Inhalt von 16 Speicherplätzen zu schreiben. Die Spalte links zeigt daher die Nummern jedes 16ten Speicherplatzes, und zwar in hexadezimaler Darstellung. Hexadezimal 0010 ist der 17te Speicherplatz; er ist in diesem Beispiel mit BF gefüllt.

Der Computer versteht nur Einsen und Nullen als Befehle. Daher ist es selbst für Spezialisten extrem schwierig ein Programm in diesem Code zu verstehen oder gar zu entwerfen. Zudem kann ein Befehl aus einem oder auch fünf Byte bestehen, und man sieht es den Bytefolgen nicht an, ob es sich um Programmcode oder Daten handelt - es gibt keinen erkennbaren Unterschied zwischen Programm- und Datencode.

Um die Programmierung von Computern zu vereinfachen, kam bald die Idee auf, Abkürzungen für die Gruppen von Einsen und Nullen zu verwenden, aus denen die Maschinensprache besteht. Die Übersetzung der Abkürzungen in Einsen und Nullen kann der Computer

selbst mit einem darauf spezialisierten Übersetzungsprogramm erledigen. Solche Programme heißen *Assembler*.

Der erste Befehl „FA“ des Beispielprogramms oben ist *clear interrupt,* auf deutsch „Unterbrechungsleitungen zeitweilig sperren“. Aus den Anfangsbuchstaben wird die Abkürzung `cli` gebildet, die für einen Programmierer leichter zu merken ist als das hexadezimale FA bzw. das binäre 11111010. Solche Abkürzungen werden mnemonische Bezeichnungen, kurz: *Mnemonics,* genannt.

Wie kommen diese Abkürzungen zustande? Der Hersteller des Prozessors, zum Beispiel Intel, liefert zu jedem neuen Prozessor ein umfangreiches Handbuch mit. Darin ist für jeden Befehl ein langer Name, ein Kurzname (Mnemonic) und eine Beschreibung des Befehls aufgeführt. Die Mnemonics sind möglichst kurz, um den Schreibaufwand beim Programmieren gering zu halten. So wird beispielsweise das englische Wort *move* (bewegen, transportieren) immer zu `mov` verkürzt und subtract (subtrahieren) zu `sub`.

Mit Mnemonics geschrieben sieht das Beispiel von oben so aus:

```
0000 FA           CLI
0001 33C0         XOR      AX,AX
0003 8ED0         MOV      SS,AX
0005 BC007C       MOV      SP,7C00
0008 8BF4         MOV      SI,SP
000A 50           PUSH     AX
000B 07           POP      ES
000C 50           PUSH     AX
000D 1F           POP      DS
000E FB           STI
000F FC           CLD
0010 BF0006       MOV      DI,0600
0013 B90001       MOV      CX,0100
0016 F2           REPNZ
0017 A5           MOVSW
0018 EA1D060000   JMP      0000:061D
001D BEBE07       MOV      SI,07BE
```

Der Befehl `cli` bedeutet wie gesagt *Clear Interrupt* („Interrupts sperren“), der darauf folgende `xor` bedeutet „exclusives oder“ (eine Logikfunktion) und bewirkt in diesem Beispiel das Löschen des AX-Registers. `mov ss, ax` heißt „Kopiere Inhalt des Registers AX nach Register SS“.

Als Register bezeichnet man die wenigen Speicherplätze, die im Prozessorkern eingebaut sind. Sie werden später erläutert.

Das vollständige Urladerprogramm finden Sie hier[1].

Moderne Assembler-Programme erleichtern dem Programmierer die Arbeit noch weiter: Sie verstehen nämlich auch symbolische Bezeichnungen, Kommentare und Organisationsanweisungen. Der Programmierer kann sie verwenden, um den Überblick im Programmcode zu erleichtern. Unser Beispiel könnte damit so aussehen:

```
        ORG     07C00             ; Startadresse des Programms festlegen
AA_MBR  SEGMENT CODE
```

1 `http://de.wikibooks.org/wiki/Maschinensprache%20i8086%2F%20Bootloader`

```
STRT:   CLI                     ; alle INTR sperren
        XOR     AX,AX
        MOV     SS,AX
        MOV     SP,Segment AA_MBR ; Sektor von 07C00 umkopieren

; Sektor mit 512 Byte ab 07C00h umkopieren nach 00600h
        MOV     SI,SP
...
        JMP     STEP2           ; Sprung nach 0000:061D

; Partitionstabelle wird nach aktiver Partition abgesucht
        ORG     0000:0600
STEP2:  MOV     SI,OFFSET PAR_TAB
...
AA_MBR  ENDS
        END
```

Die ORG-Anweisung ganz am Anfang legt fest, ab welcher Speicheradresse das Programm während der späteren Ausführung im Arbeitsspeicher untergebracht werden soll. `AA_MBR`, `STRT` und `STEP2` sind Sprungmarken, `AA_MBR` ist außerdem der Programmname. `PAR_TAB` ist die symbolische Bezeichnung der Adresse des Beginns der Partitionstabelle. Die Namen der Sprungmarken, des Programmnamens und der Adressbezeichnung hier wählt der Programmierer nach Belieben; sie könnten auch anders aussehen als in diesem Beispiel.

Das klingt kompliziert (und ist es auch). Aber wir werden im Laufe des Buches natürlich noch genauer auf die Bestandteile des Programms eingehen.

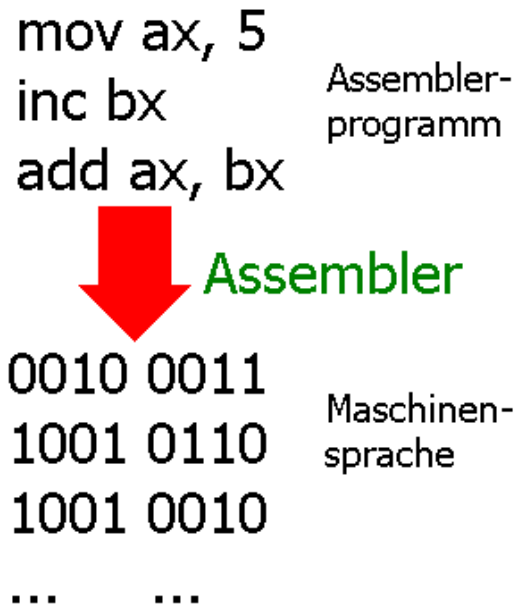

Abb. 1 Abb. 1 – Übersetzung eines Assemblerprogramms in Maschinensprache

Ein Programm, das einen solchen Quelltext in Maschinensprache übersetzen kann, nennt man wie gesagt Assemblerprogramm. Das englische Wort *to assemble* bedeutet „zusammenfügen, zusammenbauen“.

Assemblersprachen zählen zur zweiten Generation der Programmiersprachen. Ihr Nachteil: Eine Assemblersprache gilt für die Prozessoren einer Prozessorfamilie, für eine andere Familie kann sie anders lauten und der Programmierer muss umlernen.

Zur dritten Generation der Programmiersprachen zählen so genannte höhere Programmiersprachen wie Java, C#, Delphi, Visual Basic, C oder C++ u.a. Diese Sprachen sind leichter verständlich und nicht mehr auf ein bestimmtes Computersystem beschränkt: Das Programm kann man auf einem Heimcomputer, einem *Cray*-Supercomputer oder gar auf manchen Smartphones ausführen.

Angesichts der Begrenzung von Assemblersprachen auf eine Prozessorfamilie und der Unverständlichkeit des Codes liegt die Frage nahe: Wofür verwendet man heute noch Assemblersprache, wo es doch komfortable Hochsprachen wie Java, C# und andere gibt?

Einige Gründe für die Verwendung von Assemblersprache:

1. Treiber werden weitgehend in Maschinensprache geschrieben.
2. Anspruchsvolle Programme (z. B. Spiele) werden meist erst in einer Hochsprache geschrieben – und laufen unzumutbar langsam. Ersetzt man später einige kritische (am häufigsten durchlaufene) Programmteile durch Maschinensprache, wird das gesamte Programm wesentlich schneller. Ersetzt man nur ein Prozent des Programms an den richtigen Stellen durch Assemblercode, läuft das Programm 10- bis 100-mal schneller!
3. Jeder Compiler erzeugt auch aus einer Hochsprache ohnehin letztlich ein Programm in Maschinensprache.
4. Bei industriellen Steuerungen mit Echtzeit-Anforderungen, d.h. die sofort reagieren müssen, ist die Verwendung von Maschinensprache häufig notwendig.
5. Beschäftigung mit Maschinensprache erfordert und fördert zugleich das Verständnis für das Funktionieren der Hardware.

Im Gegensatz zur weit verbreiteten Meinung ist ein Betriebssystem nicht in Assembler, sondern weitgehend in einer Hochsprache geschrieben. Das liegt daran, dass der direkte Zugriff auf die Hardware nur einen geringen Teil des Betriebssystems ausmacht und in der Regel von Treibern erledigt wird. Die Aufgabe eines Betriebssystems liegt weniger in Hardwarezugriffen als vielmehr im Management der Ressourcen wie beispielsweise Management der Prozessorzeit für die Prozesse (Scheduling), Ein- / Ausgabe von Dateien und der Speicherverwaltung (Speichermanagement).

Die weitgehende Verwendung einer Hochsprache führt dazu, dass ein Kernel wie Linux (das hauptsächlich in der Hochsprache C geschrieben wurde) leicht auf viele Systeme portiert werden kann, das heißt so angepasst, dass es auf anderen Computersystemen läuft. Es müssen „nur“ die Anteile des Assemblercodes passend für die andere Prozessorfamilie umgeschrieben und der Hochsprachenanteil vom Compiler neu übersetzt werden.

1.2 Was benötige ich für die Assemblerprogrammierung?

Wenn Sie ein Assemblerprogramm für einen modernen 80x86-Prozessor erstellen möchten, benötigen Sie einen aktuellen „Assembler“ mit dem Sie die Beispielprogramme dieses Buchs in Maschinencode „assemblieren“ können. Das Buch ist passend zu dem frei erhältlichen und

als Open Source herausgegebenen Assemblerprogramm *NASM* geschrieben. Sie können es kostenlos herunterladen:

- NASM-Projektseite bei SourceForge[2]

Weitere Informationen findet man unter anderem hier:

- *The Art of Assembly*[3]
- Die Intel Pentium4-Prozessor-Dokumentation
- USENET – comp.lang.asm.x86

1.3 Welche Vorkenntnisse werden benötigt?

Assembler als Erstsprache zu erlernen ist nicht empfehlenswert. Deshalb sollten Sie bereits Grundkenntnisse in einer anderen Programmiersprache besitzen. Es wird hier nicht mehr erklärt, was eine Schleife oder eine Auswahlanweisung ist. Außerdem sollten Sie den Umgang mit Linux und Windows beherrschen.

en:X86 Assembly/Introduction[4] fr:Programmation Assembleur x86/Introduction[5] ja:X86アセンブラ/はじめに[6] nl:Programmeren in x86 assembler/Inleiding[7] pt:Assembly x86/Introdução[8]

2 http://sourceforge.net/projects/nasm
3 http://webster.cs.ucr.edu/
4 http://en.wikibooks.org/wiki/X86%20Assembly%2FIntroduction
5 http://fr.wikibooks.org/wiki/Programmation%20Assembleur%20x86%2FIntroduction
6 http://ja.wikibooks.org/wiki/X86%E3%82%A2%E3%82%BB%E3%83%B3%E3%83%96%E3%83%A9%2F%E3%81%AF%E3%81%98%E3%82%81%E3%81%AB
7 http://nl.wikibooks.org/wiki/Programmeren%20in%20x86%20assembler%2FInleiding
8 http://pt.wikibooks.org/wiki/Assembly%20x86%2FIntrodu%C3%A7%C3%A3o

2 Grundlagen

2.1 Zahlensysteme

Die Zahlensysteme dienen dazu, wie der Name andeutet, Zahlen darzustellen. Zur Darstellung von Zahlen werden in der Regel das Additions- oder das Stellenwertsystem benutzt.

Das Additionssystem kennen Sie vermutlich von den römischen Zahlen. So ist beispielsweise die Zahl DXII äquivalent mit der Zahl 512 (500 + 10 + 2). Das Additionssystem hat den Nachteil, dass insbesondere komplexe Rechenoperationen wie Multiplikation oder Division nur sehr schwer möglich sind.

Deshalb ist heute anstelle des Additionssystems praktisch nur das Stellenwertsystem gebräuchlich. Es stellt Zahlen durch eine Reihe von Ziffern z_i dar:

$$z_m z_{m-1} \dots z_0 \, , z_{-1} z_{-2} \dots z_{-n}$$

Den Zahlenwert X erhält man durch Aufsummieren aller Ziffern z_i, die mit ihrem Stellenwert b^i multipliziert werden:

$$X := \sum_{i=-n}^{m} z_i \cdot b^i$$

Dies ist eine Definition in Summenschreibweise. Die ganze Zahl i ist ein Index und wird von –n bis m durchlaufen. m ist eine natürliche Zahl und steht für den höchsten Stellenwert, -n ist eine ganze Zahl und kann 0 oder negativ sein. Ist n negativ, dann ist die Zahl gebrochen. Die natürliche Zahl b steht für die Basis des jeweiligen Stellenwertsystems. Die Summenformel kann man sich als mathematische Schreibweise einer for-Schleife denken, in der i von -n bis m durchlaufen wird und der Wert jedes Durchlaufs zu einer Gesamtsumme addiert wird -dem X hier. Wenn i den negativen Wertebereich verlässt, muss ein Komma (oder Dezimalpunkt)gesetzt werden.

Diese Definition wirkt auf viele Nichtmathematiker anfangs abschreckend. Sie zeigt aber recht übersichtlich, dass mit einem Stellenwertsystem jede Zahl als Summe dargestellt werden kann. Das ist für die folgenden Überlegungen wichtig.

Jede Ziffernposition besitzt ihren Stellenwert. Ganz rechts steht immer der niedrigste Stellenwert. Mit jeder Position weiter nach links erhöht er sich um eine Potenz.

Betrachten wir als Beispiel die Dezimalzahl 234. 4 steht hier an Stellenwert 10^0, 3 an 10^1 und 2 an Stellenwert 10^2. Somit stehen hier die einzelnen Ziffern für folgende Werte:

```
4 * 10^0 =   4
3 * 10^1 =  30
2 * 10^2 = 200
```

zusammenaddiert ergibt sich:

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

Die 10 ist hier die Basis dieses Stellenwertsystems. Die Basis ist das b der formalen Definition. Die Basis muss jedoch nicht 10 sein; jede andere ist genauso möglich. Das Stellenwertsystem mit der Basis 10 hat sich weltweit durchgesetzt, ist aber vom Prinzip her willkürlich. Woher das 10-ziffrige kommt, kann man sich vermutlich an seinen Fingern „abzählen“. Andere Zahlensysteme waren früher durchaus üblich; so benutzten beispielsweise die Babylonier 60 Ziffern (deshalb hat auch heute noch eine Stunde 60 Minuten und eine Minute 60 Sekunden).

Betrachten wir Geräte zur Datenverarbeitung auf ihrem niedrigsten Level, dem Schaltungsaufbau. Sie bestehen hauptächlich aus vielen Transistoren, die nichts anderes als ansteuerbare Schalter sind, die zwei Zustände – sperrend und durchgeschaltet – haben können. Das Dezimalsystem ist jedoch zur Betrachtung von Rechenvorgängen und Speicheradressierung, die im Grunde durch zweiwertige Schalter realisiert sind, nicht sonderlich geeignet. Besser sind hier Zahlensysteme der Basen 2^x.

2.1.1 Das Dualsystem

Da fast alle heutigen Rechner in Digitaltechnik mit zweiwertiger Logik realisiert sind, bietet sich zur anschaulichen Darstellung der internen Verarbeitungsvorgänge ein zweiwertiges Zahlensystem an, also statt der Basis 10 des Dezimalsystems ein Zahlensystem mit der Basis 2. Das zweiwertige Zahlensystem wird Dualsystem genannt und benötigt nur die Ziffern 0 und 1. Zur Unterscheidung von anderen Zahlensystemen wird oft ein b angehangen.

Nehmen wir als Beispiel die Dualzahl 111001b: Wie im Dezimalsystem hat auch hier jede Ziffer ihren Stellenwert. Während sich im Dezimalsystem der Ziffernwert mit jeder Verschiebung um eine Stelle nach links verzehnfacht, bewirkt die gleiche Verschiebung im Dualsystem eine Verdopplung. Gemäß der obigen Definition können wir die Zahl 111001b folgendermaßen darstellen, wobei die Basis nun 2 ist:

$$1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

Von links nach rechts verkörpern die einzelnen Ziffern somit folgende Werte:

$$1 \cdot 2^5 = 32$$

$$1 \cdot 2^4 = 16$$

$$1 \cdot 2^3 = 8$$

$$0 \cdot 2^2 = 0$$

$$0 \cdot 2^1 = 0$$

$$1 \cdot 2^0 = 1$$

Zusammenaddiert ergibt sich die Dezimalzahl:

$$32 + 16 + 8 + 1 = 57$$

Um ein besseres Gefühl für das Dualsystem zu bekommen, wollen wir uns die Zahlen von 0 bis 7 ansehen (in Klammer ist jeweils der Dezimalwert angegeben):

0 (0)

1 (1)

Soweit ist alles wie gewohnt. Da wir aber nun keine weitere Ziffer zu Verfügung haben, müssen wir auf die nächste Stelle ausweichen:

10 (2)

11 (3)

Um die nächst höhere Zahl darzustellen, benötigen wir wiederum eine weitere Stelle:

100 (4)

101 (5)

110 (6)

111 (7)

usw.

Dies ist sicher gewöhnungsbedürftig. Mit Übung gelingt der Umgang mit Dualzahlen aber fast so gut wie im Dezimalsystem. Auch das Rechnen mit Dualzahlen funktioniert vom Prinzip her genauso wie mit Dezimalzahlen.

Jede Ziffer bei diesem System wird als „Bit“ (Abkürzung für *binary digit*) bezeichnet. Das Bit äußerst rechts ist das *least significant bit* (lsb), das niederwertigste, und das ganz linke heißt *most significant bit* (msb), ist also das höchstwertigste.

Und wie rechnen wir eine Dezimalzahl in eine Dualzahl um?

Ganz einfach:

Die Restwertmethode

Bei der Restwertmethode wird die Dezimalzahl so oft durch zwei geteilt, bis wir den Wert 0 erhalten. Der Rest der Division entspricht von unten nach oben gelesen der Dualzahl. Sehen wir uns dies am Beispiel der Dezimalzahl 25 an:

```
25 / 2 = 12 Rest 1 (lsb)
12 / 2 =  6 Rest 0
 6 / 2 =  3 Rest 0
 3 / 2 =  1 Rest 1
 1 / 2 =  0 Rest 1 (msb)
```

Wichtig: Rechnen Sie beim Restwertsystem immer bis 0 herunter.

Als Ergebnis erhalten wir von unten nach oben gelesen die Dualzahl 11001b, was der Dezimalzahl 25 entspricht.

Bevor wir im Dualsystem weiterrechnen, folgt ein Exkurs in die Darstellung von Datenmengen und Speicherplatz: Jeweils 8 Bit ergeben ein Byte. Das Byte ist auf vielen Systemen die kleinste adressierbare Speichereinheit. 1024 Byte entsprechen wiederum einem Kilobyte. Diese auf den ersten Blick ungewöhnliche Zahl entspricht 2^{10}. Mit 10 Bit ist es möglich, 2^{10} = 1024 verschiedene Zustände darzustellen. Zur Abgrenzung zu Dezimalpräfixen sollten die von der IEC festgelegten Binärpräfixe für 2^{10}-fache verwendet werden.

Weitere häufig verwendete Größen:

Größe		**Einheit**
2^{10} Byte	(1024 Byte)	Kilobyte
2^{20} Byte	(1024 Kilobyte)	Megabyte
2^{30} Byte	(1024 Megabyte)	Gigabyte
2^{40} Byte	(1024 Gigabyte)	Terabyte

Die alte und immer noch oft zu sehende Abkürzung für 1024 Byte war KB. 1998 veröffentliche das IEC (International Electrotechnical Commission) einen neuen Standard, nachdem 1024 Byte einem Kibibyte (kilobinary Byte) entspricht und mit KiB abgekürzt wird [Nat00][1]. Damit soll in Zukunft die Verwechselungsgefahr verringert werden.

Deshalb werden in diesem Buch die neuen SI Binärpräfixeinheiten verwendet: KiB als Abkürzung für 2^{10} Byte, für 2^{20} Byte MiB, für 2^{30} Byte GiB und für 2^{40} Byte TiB.

Beispiel: Der Intel 80386 arbeitet mit einem Adressbus von 32 Bit. Wie viel Speicher kann er damit adressieren?

Lösung: Mit 32 Bit lassen sich 2^{32} verschiedene Dualzahlen darstellen, eine für jede Speicheradresse. Das heißt, der Intel 80386 kann 2^{32} = 4.294.967.296 Byte adressieren. Dies

1 Kapitel 11 auf Seite 99

sind 4.194.304 Kibibyte oder 4096 Mebibyte oder 4 Gibibyte. Verwendete man stattdessen die Dezimalpräfixeinheiten, so erhielte man: 4294.967.296 B = 4.294.967,296 KB = 4.294,967.296 MB = 4,294.967.296 GB

Nun geht es wieder an das Rechnen mit Dualzahlen.

2.1.2 Das Zweierkomplement

Wenn wir negative Zahlen im Dualsystem darstellen wollen, so gibt es keine Möglichkeit, das Vorzeichen mit + oder − darzustellen. Man könnte auf die Idee kommen, stattdessen das *Most Significant Bit* (MSB, links) als Vorzeichen zu benutzen. Ein gesetztes MSB, also mit dem Wert 1, stellt beispielsweise ein negatives Vorzeichen dar. Dann sähe zum Beispiel die Zahl −7 in einer 8-Bit-Darstellung so aus: 10000111

Diese Darstellungsform hat noch zwei Probleme: Zum einen gibt es nun zwei Möglichkeiten die Zahl 0 im Dualsystem darzustellen (nämlich 00000000 und 10000000). Weil Computer Entscheidungen scheuen, straucheln sie an dieser Stelle. Das zweite Problem: Wenn man negative Zahlen addiert, wird das Ergebnis falsch.

Um das klar zu sehen, addieren wir zunächst zwei positiven Zahlen (in Klammern steht der Wert dezimal):

```
 100011 (35)
+ 10011 (19)
Ü   11
-----------
 110110 (54)
```

Die Addition im Binärsystem ist der im Dezimalsystem ähnlich: Im Dualsystem gibt 0 + 0 = 0, 0 + 1 = 1 und 1 + 0 = 1. Soweit ist die Rechnung äquivalent zum Dezimalsystem. Da nun im Dualsystem aber 1 + 1 = 10 ergibt, findet bei 1 + 1 ein Übertrag statt, ähnlich dem Übertrag bei 9 + 7 im Dezimalsystem.

In der Beispielrechnung findet ein solcher Übertrag von Bit 0 (ganz rechts) auf Bit 1 (das zweite von rechts) und von Bit 1 auf Bit 2 statt. Die Überträge sehen Sie in der Zeile Ü.

Wir führen nun eine Addition mit einer negativen Zahl genau wie eben durch und benutzen das *Most Significant Bit* als Vorzeichen:

```
    001000 ( 8)
+ 10000111 (-7)
Ü
---------------
  10001111 (-15)
```

Das Ergebnis ist offensichtlich falsch.

Wir müssen deshalb eine andere Möglichkeit finden, mit negativen Dualzahlen zu rechnen. Die Lösung ist, negative Dualzahlen als Zweierkomplement darzustellen. Um es zu bilden, werden im ersten Schritt alle Ziffern der positiven Dualzahl umgekehrt: 1 wird 0, und

umgekehrt. Dadurch entsteht das Einerkomplement. Daraus wird das Zweierkomplement, indem wir 1 addieren.

Beispiel: Für die Zahl –7 wird das Zweierkomplement gebildet:

- 00000111 (Zahl 7)
- 11111000 (Einerkomplement)
- 11111001 (Zweierkomplement = Einerkomplement + 1)

Das Addieren einer negativen Zahl ist nun problemlos:

```
  00001000 (8)
+ 11111001 (-7, das Zweierkomplement von 7)
Ü 1111
  -------------
  00000001 (1)
```

Wie man sieht, besteht die Subtraktion zweier Dualzahlen aus einer Addition mit Hilfe des Zweierkomplements. Weiteres Beispiel: Um das Ergebnis von 35 – 7 zu ermitteln, rechnen wir 35 + (–7), wobei wir einfach für die Zahl –7 das Zweierkomplement von 7 schreiben.

Eine schwer zu findende Fehlerquelle bei der Programmierung lauert hier: Beim Rechnen mit negativen Zahlen kann es nämlich zum Überlauf (engl. Overflow) kommen, d.h. das Ergebnis ist größer als die höchstens darstellbare Zahl. Bei einer 8-Bit-Zahl etwa, deren linkes Bit das Vorzeichen ist, lassen sich nur Zahlen zwischen –128 und +127 darstellen. Hier werden 100 und 50 addiert, das Ergebnis müsste eigentlich 150 sein, aber:

```
  01100100 (100)
+ 00110010 ( 50)
Ü 11
----------------
  10010110 (-106, falls das Programm das als vorzeichenbehaftete Zahl ansieht)
```

Das korrekte Ergebnis 150 liegt nicht mehr zwischen –128 und +127. Wenn wir das Ergebnis als vorzeichenbehaftete Zahl, das heißt als Zweierkomplement, ansehen, beziehungsweise, wenn der Computer das tut, lautet es –106. Nur wenn wir das Ergebnis als positive Binärzahl ohne Vorzeichen ansehen, lautet es 150. Unser Programm muss den Computer wissen lassen, ob er das Ergebnis als Zweierkomplement oder als vorzeichenlose Binärzahl ansehen muss. Dieses Beispiel legt nahe, dass die Kenntnis dieser Grundlagen unentbehrlich ist. Andernfalls verbringt man Nächte mit erfolgloser Fehlersuche.

2.1.3 Das Oktalsystem

Mit Dualzahlen lassen sich zwar sehr gut Rechen- und Schaltvorgänge darstellen, große Zahlen benötigen jedoch viele Stellen und sind damit sehr unübersichtlich. Kompakter ist da das Oktalsystem. Seine Basis ist 2^3. Somit werden 8 unterscheidbare Ziffern benötigt, genommen werden 0 bis 7. Jede Oktalziffer kann drei Dualziffern ersetzen:

```
dual oktal
000   0
001   1
010   2
011   3
100   4
101   5
110   6
111   7
```

Oktalzahlen werden oft durch eine vorangestellte 0 kenntlich gemacht: 011 beispielsweise verkörpert die dezimale 9. Die Umrechnung von Dezimal- in Oktalzahlen geht wieder mit Hilfe der Restwertmethode:

```
234 / 8 = 29 Rest 2  Stellenwert: 8^0
 29 / 8 =  3 Rest 5      ...      8^1
  3 / 8 =  0 Rest 3      ...      8^2
```

$$234d = 0352$$

Und so geht's von Oktal- nach Dezimalzahlen: 0352 =

```
2 * 8^0 =   2
5 * 8^1 =  40
3 * 8^2 = 192
-------------
Summe:    234
```

$3 \cdot 8^2 + 5 \cdot 8^1 + 2 \cdot 8^0 = 192 + 40 + 2 = 234d$

Das Oktalsystem hat jedoch den Nachteil, dass jede Ziffer 3 Bit ersetzt, die kleinste adressierbare Speichereinheit in der Regel jedoch das Byte mit 8 Bit Speicherinhalt ist. Somit müsste man drei Oktalziffern zur Darstellung eines Bytewertes verwenden und hätte zusätzlich noch einen Bereich mit ungültigen Zahlenwerten. Besser eignet sich hierzu das Hexadezimalsystem.

2.1.4 Das Hexadezimalsystem (Sedezimalsystem)

Das Hexadezimalsystem ist ein Stellenwertsystem mit der Basis 2^4, benötigt also 16 unterscheidbare Ziffern. Jede der 2^4 Ziffern kann eine 4-stellige Dualzahl ersetzen. Da jedem Bit eine Dualziffer zugeordnet werden kann und jede Hexadezimalziffer 4-stellige Dualzahl ersetzt, kann jeder Bytewert durch eine zweistellige Hexadezimalzahl dargestellt werden.

Da das Hexadezimalsystem 16 unterscheidbare Ziffern benötigt, werden außer den Ziffern 0 bis 9 noch die Buchstaben A bis F wie in der folgenden Tabelle genutzt. Zur Unterscheidung haben Hexadezimalzahlen oft das Präfix 0x, x oder die Endung h (z. B. 0x11, x11 oder 11h).

```
dezimal binär hex okt
   0    0000   0   0
   1    0001   1   1
   2    0010   2   2
   3    0011   3   3
   4    0100   4   4
   5    0101   5   5
   6    0110   6   6
   7    0111   7   7
   8    1000   8  10
   9    1001   9  11
  10    1010   A  12
  11    1011   B  13
  12    1100   C  14
  13    1101   D  15
  14    1110   E  16
  15    1111   F  17
```

Nun rechnen wir eine Hexadezimalzahl in eine Dezimalzahl um:

$$2bd =$$

```
13 * 16^0 =  13
11 * 16^1 = 176
 2 * 16^2 = 512
---------------
Summe:      701
```

$[2BD]_{16} = 2 \cdot 16^2 + B \cdot 16^1 + D \cdot 16^0$

$2 \cdot 16^2 = 512$

$11 \cdot 16^1 = 176$

$13 \cdot 16^0 = 13$

$\sum = 701$

Die Umrechnung vom Dualsystem in das Hexadezimalsystem ist sehr einfach: Die Dualzahl wird dabei rechts beginnend in Viererblöcke unterteilt und dann blockweise umgerechnet. Beispiel: Die Zahl 1010111101b soll in die Hexadezimalform umgerechnet werden.

Zunächst teilen wir die Zahl in 4er-Blöcke ein:

10 1011 1101

Anschließend rechnen wir die Blöcke einzeln um:

10b = 2, 1011b = B, 1101b = D

somit ist

$$[1010111101]_2 = [2BD]_{16}$$

Wichtig: Die Viererblöcke müssen von „hinten“, also vom *least significant bit* (lsb) her, abgeteilt werden.

Stolperfalle beim Programmieren später: Einige Prozessoren schreiben das *least significant bit* rechts, andere links. Das ist noch nicht wichtig, aber wer schon jetzt nachschlagen will, suche die Stichwörter *little endian* und *big endian*.

Üblich und daher auch im Assembler NASM ist die Schreibweise von Hexadezimalzahlen mit einem vorangestellten ‚0x‘ (z. B. 0x2DB) oder mit der Endung ‚h‘ (z. B. 2DBh). Wir verwenden die Schreibweise mit ‚h‘.

2.2 Der ASCII-Zeichensatz und Unicode

Ein Rechner speichert alle Informationen numerisch, das heißt als Zahlen. Um auch Zeichen wie Buchstaben und Satzzeichen speichern zu können, muss er jedem Zeichen einen eindeutigen Zahlenwert geben.

Die ersten Kodierungen von Zeichen für Computer gehen auf die Hollerith-Lochkarten (benannt nach deren Entwickler Herman Hollerith) zurück, die zur Volkszählung in den USA 1890 entwickelt wurden. Auf diesen Code aufbauend entwickelte IBM den 6-Bit-BCDIC-Code *(Binary Coded Decimal Interchange Code),* der dann zum EBCDIC *(Extended Binary Coded Decimal Interchange Code),* einem 8-Bit-Code, erweitert wurde.

1968 wurde der *American Standard Code for Information Interchange* (ASCII) verabschiedet, der auf einem 7-Bit-Code basiert und sich schnell gegen andere Standards wie dem EBCDIC durchsetzte. Das achte Bit des Bytes wurde als Prüf- oder Parity-Bit zur Sicherstellung korrekter Übertragungen verwendet.

Im Unterschied zum heute praktisch nicht mehr verwendeten, konkurrierenden EBCDI-Code gehört beim ASCII-Code zu jedem möglichen Binärzeichen ein Zeichen. Das macht den Code ökonomischer. Außerdem liegen die Groß- und die Kleinbuchstaben jeweils in zusammenhängenden Bereichen, was für die Programmierung deutliche Vorteile hat, mit einem Pferdefuß: Werden Strings (Zeichenfolgen, wie etwa Wörter) nach ASCII sortiert, geraten die klein geschriebenen Wörter hinter alle groß beginnenden. Die lexikalische Sortierung erfordert daher etwas mehr Aufwand.

Neben den alphabetischen Zeichen in Groß- und Kleinbuchstaben erhält der ASCII-Code noch eine Reihe von Sonderzeichen und Steuerzeichen. Steuerzeichen sind nicht druckbare Zeichen, die ursprünglich dazu dienten, Drucker oder andere Ausgabegeräte zu steuern. Die 128 Zeichen des ASCII-Codes lassen sich grob so einteilen (in Klammern die Nummern in hexadezimaler Notierung):

- Zeichen 0 bis 31 (1Fh): Steuerzeichen, wie zum Beispiel Tabulatoren, Zeilen- und Seitenvorschub
- Zeichen 48 (30h) bis 57 (39h): Ziffern 0 bis 9
- Zeichen 65 (41h) bis 90 (5Ah): Großbuchstaben A bis Z
- Zeichen 97 (61h) bis 122 (7Ah): Kleinbuchstaben a bis z

Leer- und Interpunktionszeichen, Klammern, Dollarzeichen usw. verteilen sich in den Zwischenräumen.

Bei diesem Standard fehlen allerdings Sonderzeichen wie beispielsweise die Umlaute. Mit dem Erscheinen des PCs erweiterte IBM den ASCII-Code auf 8 Bit, so dass nun 128 weitere Zeichen darstellbar waren. Neben einigen Sonderzeichen besaß der "erweiterte ASCII-Code" auch eine Reihe von Zeichen zur Darstellung von Blockgrafiken.

Mit MS-DOS 3.3 führte Microsoft die sogenannten *„Code Pages“* ein, die den Zeichen von 128 bis 255 eine flexible auf den jeweiligen Zeichensatz angepasst werden konnte. Damit konnte beispielsweise auch der griechische Zeichensatz dargestellt werden.

Das eigentliche Problem lösten die *Code Pages* indes nicht: die Verwendung unterschiedlicher *Code Pages* machte zwar die für die europäischen Sprachen benötigten Zeichen verfügbar, führt aber bis heute dazu, dass zum Beispiel Umlaute in E-Mails nicht korrekt übertragen werden, falls das sendende Mail-Programm nicht korrekt konfiguriert oder fehlerhaft programmiert ist. Noch problematischer war, dass japanische oder chinesische Zeichen nicht dargestellt werden konnten, da ihre Anzahl die 256 möglichen Zeichen bei weitem überstieg. Dazu wurden DBCS *(double byte character set)* eingeführt, der den Zeichensatz auf 16 Bit erweitert, allerdings auch noch den 8 Bit ASCII-Zeichensatz unterstützt.

Mit dem Unicode schließlich wurde ein Schlussstrich unter die Wirrungen in Folge der verschiedenen Erweiterungen gezogen und ein einheitlicher 16-Bit-Zeichensatz eingeführt. Mit Unicode ist es möglich, 65536 verschiedene Zeichen und sämtliche vorhandenen Sprachen in einem Zeichensatz darzustellen.

3 Der Prozessor

3.1 Die Von-Neumann-Architektur

Fast alle heute entwickelten Rechner basieren auf der Von-Neumann-Architektur. Sie stammt von John von Neumann, der 1946 in einem Konzept skizzierte, wie ein Universalrechner aussehen sollte ([Sie04][1]):

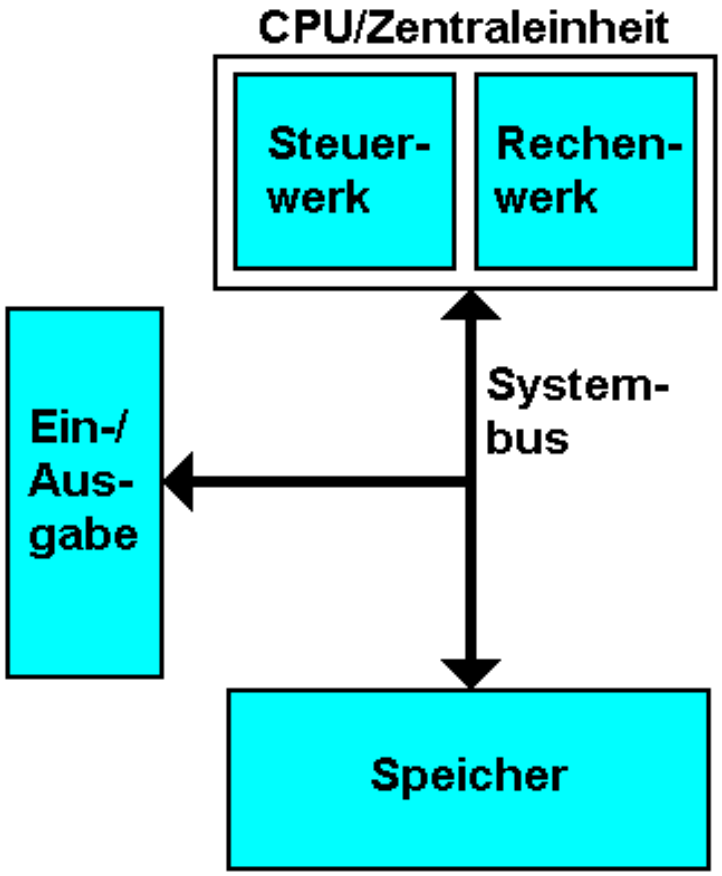

Abb. 2 Von Neumann Architektur

- Der Aufbau des Rechners ist unabhängig von der Problemstellung (daher auch die Bezeichnung Universalrechner). Durch die Programmierung wird er in die Lage versetzt, wechselnde Aufgaben zu lösen.
- Jeder Rechner besteht demnach mindestens aus den folgenden Komponenten:

1 Kapitel 11 auf Seite 99

1. Der Zentraleinheit bzw. der CPU *(**C**entral **P**rocessing **U**nit).* Die CPU wiederum besteht aus dem Steuerwerk und dem Rechenwerk.
2. Dem les- und beschreibbaren Speicher
3. Den Ein-/Ausgabeeinheiten
4. Einem Systembus, der alle Einheiten miteinander verbindet

- Programme und Daten werden im selben schreib- und lesbaren Speicher abgelegt, Anpassungen des Programms sind dadurch immer möglich. Der Speicher ist dabei fortlaufend durchnummeriert, so dass jede Speicherzelle über eine Nummer (man sagt auch: ihre Adresse) angesprochen werden kann.
- Programmausführung, Lesen und Schreiben von Daten und Kommunikation der Ein-/Ausgabeeinheiten verwenden den gleichen Systembus. Dadurch ist sichergestellt, dass ein Programmablauf deterministisch jedes mal zum gleichen Ergebnis führt (im Gegensatz zu z. B. Architekturen die getrennte Daten- und Programmbusse besitzen).

Der große Vorteil der Von-Neumann-Architektur ist die strikte Sequentialität der Ausführung und der Datenzugriffe. Dadurch kann sich der Programmierer auf deterministisches Verhalten seines Programms verlassen. Sie hat jedoch vor allem einen Nachteil: Der Bus ist der Flaschenhals des Systems. Moderne Prozessoren sind in der Regel wesentlich schneller bei der Ausführung als die Daten, die über den Bus vom Speicher in den Prozessor geladen werden können. Als Folge davon wird die Programmausführung gebremst. Man spricht in diesem Zusammenhang deshalb auch vom „Von-Neumannschen-Flaschenhals“ (engl. The von Neumann Bottleneck). Daher versuchen heutige Prozessoren, die Bearbeitung von Programmen zu parallelisieren, anstatt sie sequenziell abzuarbeiten, sowie Daten und Code teilweise in unterschiedlichen Speichern zu halten (Cache). Damit weichen sie immer mehr vom Von-Neumann-Konzept ab, das Programmausführungsresultat entspricht aber weiterhin dem einer strikt sequentiellen Architektur. Da diese Abweichungen das Erlernen der Assemblerprogrammierung nur in geringem Maße beeinflussen, werden wir im Folgenden nur am Rande auf diese Änderungen eingehen.

3.2 Die 80x86-Prozessorfamilie

Bevor wir uns eingehender mit der CPU *(Central Processor Unit)* beschäftigen, wollen wir uns einen Überblick über die Entwicklung der Intelprozessoren verschaffen. Interessant ist, dass alle aktuellen Prozessoren abwärtskompatibel sind. Das heißt, selbst ein moderner Prozessor wie der Pentium 4 kann sich verhalten wie ein 8086-Prozessor – freilich nutzt er dann nur einen kleinen Teil seiner Fähigkeiten. Der Nachteil daran ist, dass alle 80x86-CPUs heute alte Technik enthalten. Es lohnt ein Blick auf die Anfangszeiten, um die Zusammenhänge zu verstehen.

8088, 8086

Der 8086 ist eine 16-Bit-CPU und gehörte im Jahr seines Erscheinens 1978 zu den leistungsfähigsten Prozessoren für Personal Computer. Er konnte ein Megabyte Speicher adressieren (20 Bit breiter Adressbus) und arbeitete im sogenannten *Real Mode.* In diesem Modus konnte ein Programm auf jede verfügbare Adresse im Speicher zugreifen – eingeschlossen den Code und die Daten anderer Programme. Der Speicher von Programmen wurde in Segmente aufgeteilt, die höchstens 64 Kb einnehmen konnten.

Allerdings war das Mainboard für die 8086-CPU sehr teuer, weshalb Intel sich entschloss, eine Billigvariante zu entwickeln: die 8088-CPU. Sie war zwar ebenfalls eine 16-Bit-CPU, besaß allerdings nur einen Datenbus von 8 Bit Breite. Für das Programmieren war dies aber nicht relevant, da dieser Prozessor 16-Bit-Zugriffe selbstständig auf jeweils zwei 8-Bit-Zugriffe verteilt, was eben dementsprechend langsamer ist.

80186, 80188

Der Vollständigkeit halber sollen hier noch kurz die Nachfolger dieser beiden Prozessoren erwähnt werden. Deren Befehlsumfang wurde erweitert und stellenweise optimiert, so dass ein 80186 etwa ein Viertel schneller arbeitet als ein gleich getakteter 8086. Der 80186 ist eher ein integrierter Mikrocontroller als ein Mikroprozessor, denn er enthält auch einen *Interrupt Controller,* einen DMA-Chip und einen Timer. Diese waren allerdings inkompatibel zu den vorher verwendeten externen Logiken des PC/XT, was eine Verwendung dort sehr schwierig machte. Darum wurde er im PC selten verwendet, war jedoch auf Adapterkarten als selbstständiger Mikroprozessor relativ häufig zu finden, beispielsweise auf ISDN-Karten.

Der 80188 ist wiederum eine abgespeckte Version mit 8-Bit-Datenbus, der 16-Bit-Zugriffe selbsttätig aufteilt. Wie beim 80188 sind 8-Bit-Zugriffe genauso schnell wie beim 80186, 16-Bit-Zugriffe benötigen die doppelte Zeit.

80286

Man kann den 80286 als Nachfolger des 8086 sehen, da der 80186/80188 durch die Inkompatiblitäten nicht weit verbreitet war. Das interessanteste neue Feature war der 16 Bit *Protected Mode*. Damit ist es möglich, Speicher, der von anderen Programmen reserviert ist, zu schützen. Das ist eine Voraussetzung für Multitasking-Betriebssysteme, die mehrere Programme zugleich in den Arbeitsspeicher laden können. Im Protected Mode konnten gegenüber dem 1 MB adressierbaren Arbeitsspeicher bis zu 16 MB adressiert werden.

80386

Mit dem 80386 entwickelte Intel erstmals eine 32-Bit-CPU. Damit konnte der Prozessor 4 GB Speicher linear ansprechen. Da sich der *Protected Mode* anfänglich nicht durchsetzen konnte, enthielt die 80386-CPU einen virtuellen 8086-Modus. Er kann aus dem *Protected Mode* heraus gestartet werden und simuliert einen oder mehrere 8086-Prozessoren. Die internen Speichereinheiten des Prozessor, die Register, wurden von 16 Bit auf 32 Bit verbreitert.

Wie schon beim 8086 brachte Intel mit dem 80386SX eine *Low-Cost*-Variante in den Handel, der statt des 32 Bit breiten nur einen 16 Bit breiten Datenbus hatte.

80486

Mit dem 80486-Prozessor brachte Intel eine überarbeitete 80386-CPU auf den Markt, die um einen Cache und den mathematischen Coprozessor erweitert wurde, der b-Chip. Er ist teurer, aber viel schneller als der Hauptspeicher. Der schnelle Cache kann die Ausführung der Programme beschleunigen, da die CPU nur noch selten auf den langsamen Hauptspeicher zugreifen muss. Das ist dem Lokalitätsprinzip geschuldet. Es besagt, dass die Ausführung sich im Wesentlichen auf einen kleinen Bereich des Codes und der Daten beschränkt. So findet der Prozessor meist im Cache, was er gerade braucht.

Der mathematische Coprozessor dient dazu, komplexere mathematische Operationen auszuführen und führte Unterstützung für Gleitkommadatentypen ein. Auch für den anfänglich

noch teuren 80486-Prozessor gab es mit dem 80486SX eine „Sparvariante“, die ohne den mathematischen Coprozessor arbeitete. Der Datenbus wurde dabei nicht beschränkt. Mit seinem Cache war er eigentlich nur eine schnellere 80386-CPU.

80586/Pentium-Architektur Mit dem Pentium-Prozessor (und den kompatiblen Varianten der Konkurrenz) wurde unter anderem ein Konzept der parallelen Instruktionsausführung in die Prozessoren aufgenommen, die superskalare Ausführung. Der Prozessor analysiert die Befehle die demnächst auszuführen sind. Falls keine direkte Abhängigkeit der Instruktionen untereinander besteht und die nötige Ressource (z. B. der Coprozessor) frei ist, wird die Ausführung vorgezogen und die Instruktion parallel ausgeführt. Dies verbessert die Ausführungsgeschwindigkeit deutlich, macht jedoch auch das Assembler Programmieren komplexer, denn die Ausführungszeit von Instruktionen hängt nun vom Kontext der vorher und nachher auszuführenden Instruktionen ab.

Mit der späteren MMX-Erweiterung der Pentium-Prozessor-Architektur fand eine weitere Form der parallelen Ausführung seinen Platz in der x86-Architektur, die parallele Datenverarbeitung auf Vektoren (SIMD). Hierzu wurden der Befehlssatz um Vektoroperationen erweitert, welche mit den Registern des Coprozessor verwendet werden können.

x86-64-Architektur Mit der x86-64-Architekturerweiterung, eingeführt von AMD und später von Intel übernommen, wurden die bisherigen Konzepte und Instruktionen der 32-Bit-x86-Prozessorarchitektur beibehalten. Die Register und Busse wurden von 32 Bit auf 64 Bit erweitert und ein neuer 64Bit Ausführungsmodus eingeführt, die 32-Bit-Betriebsmodi sind weiterhin möglich. Allerdings fiel hier der *virtual protected Mode* weg, der eh nicht mehr verwendet wurde.

3.3 Die Register der 8086/8088-CPU

Die CPU besitzt eigene interne Speicher, die sogenannten Register. Auf die Daten in den Registern kann die CPU viel schneller zugreifen als auf die Daten im Hauptspeicher oder im Cache. Allerdings bieten die Register noch weniger Speicherplatz als der Cache: Nur 14 Register, die je 16 Bit aufnehmen können, finden sich auf dem Chip der 8086-CPU. Daher muss der Assembler-Programmierer sehr darauf achten, dass sich nur die nötigsten Daten in den Registern befinden.

Die Register können in Typen unterteilt werden:

- Datenregister sind in der Regel universell einsetzbar und nehmen Operanden auf.
- Adressregister enthalten keine Daten, sondern die Adresse einer Speicherzelle. Sie sind nötig, um Adressen zu berechnen (man spricht hier auch von Adressarithmetik).
- Stackregister: Es dient zur Verwaltung des Stacks. Der Stack ist eine Datenstruktur, auf der der Inhalt einzelner Register wie auf einem Stapel vorübergehend abgelegt werden kann.
- Spezialregister: Die benötigt der Prozessor für sich. Der Programmierer kann sie meist nur indirekt ändern.

Die Register im Einzelnen:

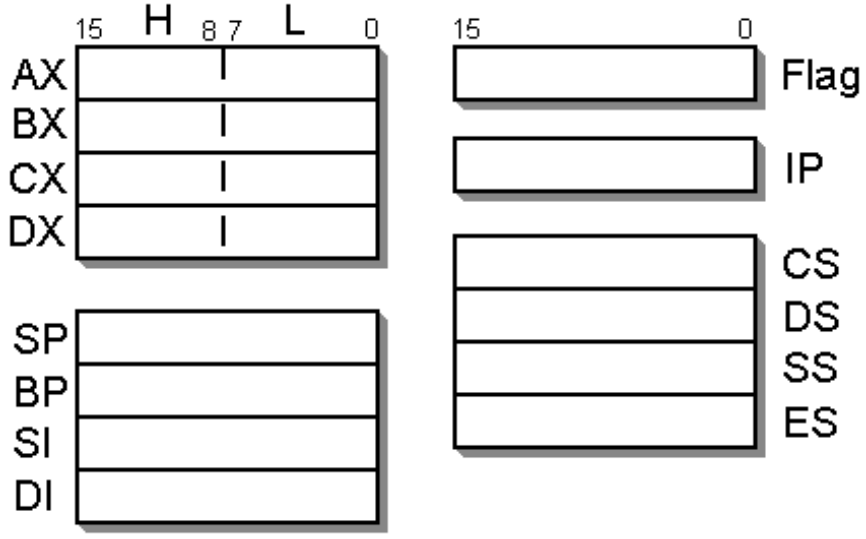

Abb. 3 Registersatz der 8088/86-CPU

Datenregister

Das AX-Register

Das AX-Register (Akkumulator) dient hauptsächlich der Durchführung von Rechenoperationen. Einige Arithmetikbefehle laufen ausschließlich über dieses Register.

Das BX-Register

Dieses Register wird häufig bei indirekten Speicheroperationen genutzt.

Das CX-Register

Das CX-Register *(Count Register)* dient einigen Maschinenbefehlen als Zählregister. Beispielsweise legt der Schleifenbefehl die Anzahl der Durchgänge im CX-Register ab.

Das DX-Register

Das DX-Register unterstützt das AX-Register bei der Berechnung von Zahlen, deren Ergebnis länger als 16 Bit ist.

Diese Register fungieren auch als allgemeine Register, solange sie nicht für besondere Aufgaben gebraucht werden.

Die unteren und die oberen 8 Bit der Register AX, BX, CX und DX lassen sich getrennt ansprechen. Mit AL (Low) werden beispielsweise die unteren 8 Bit des AX Registers angesprochen, mit AH (High) die oberen acht.

Adressregister

CS-, DS-, ES- und SS-Register

Eine spezielle Bedeutung unter den Registern haben die Segmentregister CS (Codsegmentregister), DS (Datensegmentregister), ES (Extrasegmentregister) und SS (Stacksegmentregister). Sie bilden im so genannten *Real Mode* gemeinsam mit dem Offset die physikalische

Adresse. Man kann sie nur dafür nutzen; sie können weder als allgemeine Register benutzt, noch direkt verändert werden.

Stackregister

BP- und SP-Register

Während das *Base-Pointer*-Register (BP) meistens als allgemeines Register benutzt werden kann und nur zur Adressberechnung herangezogen wird, dient das *Stack-Pointer*-Register (SP) zusammen mit dem Stacksegmentregister zur Verwaltung des Stacks. Der Stack ist ein Bereich im Arbeitsspeicher, in dem Werte der Register zwischengespeichert werden können. Der Stack funktioniert nach dem sogenannten LIFO-Prinzip *(last in, first out).* Das bedeutet, dass der Wert, der als letztes auf den Stack gelegt wurde, auch als erstes wieder von ihm geladen werden muss – ähnlich einem Stapel oder Keller.

SI- und DI-Register

Das Quellindex-Register SI *(Source Index)* und das Zielindex-Register DI *(Destination Index)* dienen zur Adressberechnung bei String-Befehlen, wenn zum Beispiel Strings innerhalb des Speichers verschoben werden müssen.

Spezialregister

Das IP-Register

Das *Instruction-Pointer*-Register (IP) enthält zusammen mit dem CS-Register immer die Adresse des Speicherplatzes mit dem als nächstes auszuführenden Befehl. Dadurch „weiß“ der Prozessor immer, wo er mit der Bearbeitung des Programms fortsetzen muss. Das IP-Register kann der Programmierer nicht mit einem anderen Wert füllen. Eine Änderung ist nur indirekt über Sprungbefehle oder Prozeduraufrufe möglich, wodurch das Programm an einer anderen Stelle fortgesetzt wird.

Das Flag-Register

Das Flag-Register ist ein Register von 16 Bits, die in diesem Register Flags heißen, von denen jedes eine Spezialaufgabe hat und einzeln mit einem Wert gefüllt werden kann. Hat eines den Wert 1, spricht man von einem gesetzten Bit, oder hier: Flag. Ist es 0, nennt man es gelöscht.

Das Flag-Register dient der Kommunikation zwischen Programm und Prozessor. So kann beispielsweise das Programm richtig reagieren, wenn unerwartet ein Ergebnis nicht in ein 16-Bit-Register passt und der Programmierer dafür gesorgt hat, dass das Programm das dafür zuständige Flag prüft. Der Prozessor und das Programm können Flags setzen oder zurücksetzen.

Die Flags im Einzelnen:

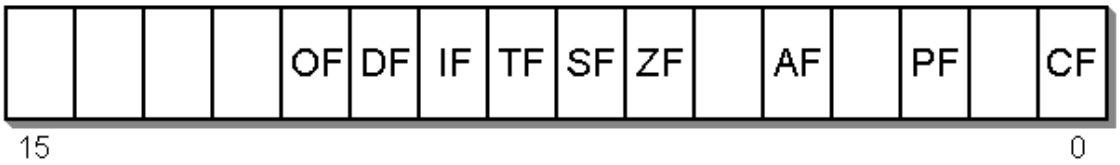

Abb. 4

- Bit 11 – Überlaufflag / *Overflow Flag* (OF)
- Bit 10 – Richtungsflag / Direction Flag*(DF)*
- Bit 9 – *Interruptflag* (IF)
- Bit 8 – Einzelschrittflag / *Trap Flag* (TF)
- Bit 7 – Vorzeichenflag / *Sign Flag* (SF)
- Bit 6 – Nullflag / *Zero Flag* (ZF)
- Bit 4 – *Auxiliary Flag* (AF)
- Bit 2 – Paritätsflag / *Parity Flag* (PF)
- Bit 0 – Übertragsflag / *Carryflag* (CF)

Einige der Flags haben bei der 8086-CPU noch keine Bedeutung und sind für spätere CPUs reserviert. Deshalb darf der Programmierer sie nicht verwenden.

3.4 Bearbeitung eines Maschinenprogramms

Jedes Programm besteht aus einer Reihe von Befehlen, die der Prozessor bearbeitet. In einem vereinfachten Modell verläuft die Ausführung eines Maschinenprogramms in zwei Phasen:

1. Der Programmzähler wird auf die Adresse des Speicherplatzes gesetzt, in dem sich der Code des nächsten Befehls befindet. Dann wird der Befehlscode aus dem Arbeitsspeicher geholt und dekodiert (Fetchphase).
2. Der Befehl wird ausgeführt (Ausführungsphase).

Danach wird der Programmzähler erhöht und ein neuer Zyklus beginnt.

Dies ist allerdings ein idealisiertes Bild, weil eine ganze Reihe von Ausnahmen dazu führen kann, dass der Prozessor die Programmausführung unterbricht:

- Der Maschinenbefehl braucht einen zusätzlichen Operanden aus einer Speicherzelle und benötigt dafür zusätzliche Lesezyklen.
- Der Prozessor muss einen Sprung- oder einen Schleifenbefehl ausführen und den Programmzähler mit der Adresse des Speicherplatzes laden, der den Befehl am Sprungziel enthält. Auch beim Ausführen eines Unterprogramms muss der Programmzähler neu geladen werden.
- Der Prozessor empfängt ein Interrupt-Signal. Es unterbricht das Programm, damit die CPU ein Spezialprogramm, eine Interruptroutine, ausführt, etwa zur Beseitigung einer Störung (mehr dazu unter Interrupts[2]).

2 Kapitel 3.7 auf Seite 27

3.5 BIU und EU

Früher musste die CPU nach jedem Befehl den nächsten Befehl aus dem Arbeitsspeicher holen. Das kostete Zeit, während der die CPU in der Abarbeitung der Befehle unterbrochen wurde. Um den Zeitverlust zu mindern, parallelisieren die Entwickler heute die Vorgänge des Holens und des Ausführens, das heißt sie erledigen beides gleichzeitig. Dazu besitzt die CPU zwei getrennte Funktionseinheiten:

- Die *Executive Unit* (EU) und
- die *Bus Interface Unit* (BIU)

Die BIU ist für die Verbindung von Prozessor und Bussystem verantwortlich, die EU führt die Befehle aus. Beide sind unabhängig voneinander und arbeiten parallel.

Während der Ausführung eines Maschinenprogramms lädt die BIU den nächsten Befehl aus dem Arbeitsspeicher in eine interne Befehlswarteschlange, den sogenannten *Prefetch Buffer.* Von dort holt ihn sich die EU und bearbeitet ihn. Zugleich lädt die BIU den darauf folgenden Befehl in den *Prefetch Buffer.*

Moderne Prozessoren parallelisieren sogar die Arbeit der *Executive Unit.* Diese Technik bezeichnet man als „superskalar“. Ziel ist abermals die schnellere Programmausführung.

3.6 Der Bus

Wie wir gesehen haben, werden im Von-Neumann-Rechner alle Komponenten mit dem Systembus verbunden. Der Systembus lässt sich in drei Teile gliedern:

Der Datenbus

Der Datenbus ist für die Übertragung von Daten und Programmen zwischen den Komponenten des Rechners zuständig. Der Prozessor kann auf den Datenbus in zwei Richtungen zugreifen: lesend und schreibend. So eine Verbindung, die in beide Richtungen möglich ist, bezeichnet man auch als bidirektional.

Der Adressbus

Der Adressbus dient dem Prozessor zum Ansprechen des Arbeitsspeichers. Durch ihn gelangt der Prozessor an den Inhalt einer Speicherzelle, indem er die Adresse über den Adressbus schickt, man sagt auch: auf den Adressbus legt. Anders als der Datenbus leitet der Adressbus Informationen nur in eine Richtung, nämlich vom Steuerwerk des Prozessors zum Arbeitsspeicher. Er wird daher als unidirektional bezeichnet.

Die Breite des Adressbusses, das heißt die Anzahl seiner Datenleitungen, bestimmt, wie viele Speicherzellen des Arbeitsspeichers er höchstens adressieren kann. Die 8086/8088-CPU beispielsweise besitzt einen 20 Bit breiten Adressbus. Damit kann sie auf $2^{20} = 1$ MB Speicher zugreifen.

Der Kontroll- bzw. Steuerbus

Der Kontrollbus koordiniert den Adress- und den Datenbus. Er sorgt dafür, dass nicht mehrere Komponenten gleichzeitig darauf zugreifen.

3.7 Interrupts

Während die CPU ein Programm bearbeitet, können verschiedene Ereignisse wie diese eintreten: Der Anwender drückt eine Taste, dem Drucker geht das Papier aus oder die Festplatte ist mit dem Schreiben der Daten fertig.

Um darauf reagieren zu können, hat der Prozessor zwei Möglichkeiten:

- Die CPU kann ständig nachfragen, ob so ein Ereignis eingetreten ist. Dieses Verfahren bezeichnet man als programmierte I/O oder auch Polling. Nachteil dieser Methode ist, dass die ständige Prüfung den Prozessor arg aufhält.
- Das Gerät erzeugt eine Nachricht, sobald ein Ereignis eintritt, und schickt diese über einen Interrupt-Controller an den Interrupteingang des Prozessors. Die CPU unterbricht das gerade laufende Programm und reagiert auf das Ereignis. Anschließend kann sie mit der unterbrochenen Arbeit fortfahren.

Ein solches Interrupt-Signal, das von einem Gerät zur CPU geschickt wird, bezeichnet man im Unterschied zu Software-Interrupts als Hardware-Interrupt. Hardware-Interrupts können asynchron auftreten, d.h. es ist nicht vorhersagbar, wann einer aufritt. Einen Softwareinterrupt hingegen löst die Anwendungssoftware aus, bevor sie auf Systemfunktionen des Betriebssystems zugreifen kann. Software-Interrupts treten im Unterschied zu Hardware-Interrupts synchron auf, d.h. vorhersagbar, weil das im Programm so festgelegt ist.

Manchmal ist es nötig, die Unterbrechung des Programms durch Interrupts zu verhindern, zum Beispiel bei der Ausnahmebehandlung. Dazu löscht man das Interruptflag (IF = 0) mit dem Befehl `cli` *(Clear Interruptflag)*. `sti` *(Set Interruptflag)* setzt es (IF = 1). Sind die Interrupts auf diese Weise gesperrt, kann das System auf keine Ein- oder Ausgabe mehr reagieren. Der Anwender merkt das daran, dass das System seine Eingaben über die Tastatur oder die Maus nicht mehr bearbeitet.

Die 80x86-CPU hat zwei Interrupt-Eingänge, den INTR *(Interrupt Request)* und den NMI *(Non Maskable Interrupt)*. Der Interrupt INTR ist für normale Anfragen an den Prozessor zuständig, während der NMI nur besonders wichtigen Interrupts vorbehalten ist. Das Besondere am *Non Maskable Interrupt* („nicht maskierbar“) ist, dass er nicht durch Löschen des Interruptflags unterdrückt werden kann.

3.8 Adressierung des Arbeitsspeichers im *Real Mode*

Der Adressbus dient der CPU dazu, Speicherplätze im Arbeitsspeicher anzusprechen, indem sie die Adresse eines Speicherplatzes über die Leitungen des Busses sendet. Die 8086/88-CPU hat dafür 20 Adressleitungen, die den Adressbus bilden. Damit könnte der Prozessor unmittelbar $2^{20} = 1$ MByte adressieren, wenn auch das zuständige Register in der CPU 20 Bit breit wäre. In dem Register muss sich die Adresse befinden, bevor sie auf den Adressbus geht. Mit ihren 16-Bit-Registern kann die 8086/88-CPU jedoch höchstens $2^{16} = 65.536$ Byte direkt adressieren.

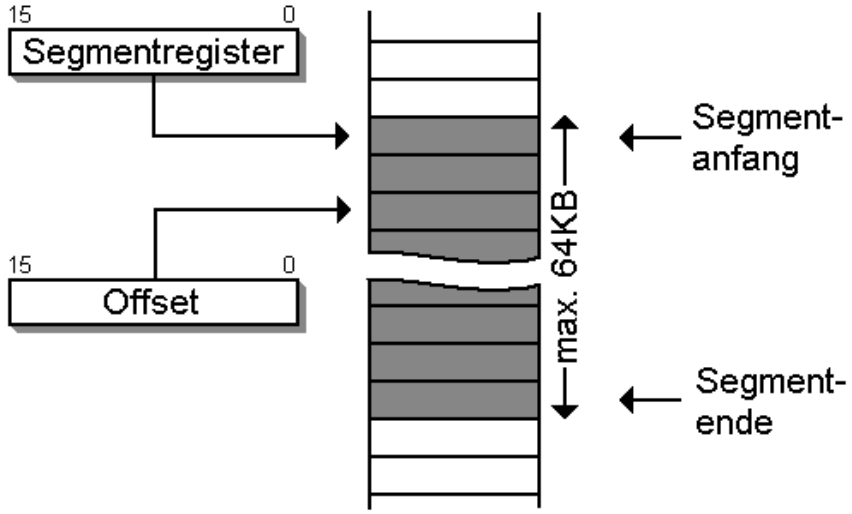

Abb. 5 Segmentierung in *Real Mode*

Um mit dieser CPU ein Megabyte Speicher anzusprechen, benötigt man zwei Register. Mit zwei 16-Bit-Registern können wir $2^{32} = 4$ Gigabyte Arbeitsspeicher ansprechen - mehr als wir benötigen. Eine 20-Bit-Adresse könnte mit Hilfe der zwei Register gebildet werden, indem man auf das erste Register die ersten 16 Bit der Adresse legt und auf das zweite die restlichen 4 Bit. Ein Vorteil dieser Methode ist, dass der gesamte Adressraum linear angesprochen werden kann. „Linear“ bedeutet, dass die logische Adresse – also die, die sich in den Registern befindet – mit der physikalischen Adresse der Speicherzelle im Arbeitsspeicher übereinstimmt. Das erspart uns die Umrechnung zwischen logischer und physikalischer Adresse.

Das Verfahren hat allerdings auch Nachteile: Die CPU muss beim Adressieren immer zwei Register laden. Zudem bleiben zwölf der sechzehn Bits im zweiten Register ungenutzt.

Deshalb ist im Real Mode ein anderes Adressierungsverfahren üblich: Mit einem 16-Bit-Register können wir 65.536 Byte an einem Stück ansprechen. Bleiben wir innerhalb dieser Grenze, brauchen wir für die Adressierung einer Speicherzelle kein zweites Register.

Um den ganzen Speicherbereich von 1 MByte zu nutzen, können wir mehrere 65.536-Byte-Blöcke über den gesamten Speicher von 1 MByte verteilen und jeden von ihnen mit dem zweiten Register adressieren. So ein Block heißt Segment. Das zweite Register legt fest, an welcher Speicheradresse ein Segment anfängt, und mit dem ersten Register bewegen wir uns innerhalb des Segments.

Der Speicher von 1 MByte, der mit den 20 Leitungen des Adressbusses adressiert werden kann, ist allerdings nicht in ein paar säuberlich getrennte Segmente unterteilt. Vielmehr überlappen sich eine Menge Segmente, weil jedes 16. Byte, verteilt über 1 MByte, als möglicher Beginn eines 64-KByte-Segments festgelegt wurde. Der Abstand von 16 Byte kommt zu Stande, weil die erlaubten Anfänge der Segmente gleichmäßig auf die 1 MByte verteilt wurden (Division 1.048.576 Byte durch 65.536).

Die Überlappung der Segmente ist eine Stolperfalle für Programmierer: Ein und dieselbe Speicherzelle im Arbeitsspeicher kann durch viele Kombinationen von Adressen des ersten

und des zweiten Registers adressiert werden. Passt man nicht auf, dann überschreibt ein Programm unerlaubt Daten und Programmcode und stürzt ab – eine so genannte Schutzverletzung ist aufgetreten.

Die Adresse im zweiten Register, die den Anfang eines Segments darstellt, heißt Segmentadresse. Die physikalische Adresse im Arbeitsspeicher errechnet sich aus der Segmentadresse und der Adresse des ersten Registers, der so genannten Offsetadresse. Erst wird die Segmentadresse mit 16, dem Abstand der Segmente, multipliziert. Zur Multiplikation mit 16 hängt man rechts einfach vier Nullen an. Hinzu wird die Offsetadresse addiert:

Physikalische Adresse = Segmentadresse * 16 + Offsetadresse

Die Offsetadresse im ersten Register erlaubt, auf ein 65.536 Byte großes zusammen hängendes Speicherfeld zuzugreifen. In Verbindung mit der Segmentadresse im zweiten Register kann man den gesamten Speicher erreichen. Beschränkt sich ein Programm auf die Speicherplätze innerhalb eines Segments, braucht die Segmentadresse gar nicht geändert werden.

Sehen wir uns dazu ein Beispiel an: Nehmen wir an, dass das Betriebssystem ein Programm lädt, das 40 KByte Daten und 50 KByte Code umfasst. Daten und Code wandern in den Arbeitsspeicher, wo sie säuberlich getrennt eigene Speicherbereiche einnehmen sollen. Um separate Speicherbereiche für die Daten und den Code zu schaffen, initialisiert das Betriebssystem zuvor das Datensegmentregister (DS) und das Codesegmentregister (CS) mit den Adressen, an denen beginnend die Daten und der Code im Arbeitsspeicher gelagert werden.

Wollen wir beispielsweise Text ausgeben, müssen wir eine Betriebssystemfunktion aufrufen, der wir die Adresse des Textes übergeben, den sie ausgeben soll. Die Segmentadresse des Textes steht schon im Datensegmentregister. Wir müssen der Betriebssystemfunktion nur noch die Offsetadresse nennen. Man sieht: Das Betriebssystem erfährt die vollständige Adressinformation, obwohl wir der Funktion nur den Offsetanteil übergeben haben.

Kommt ein kleines Programm mit weniger als 64 kByte inklusive aller Daten und des Codes aus, braucht man das Segmentregister beim Programmstart nur einmal zu initialisieren. Die Daten erreicht man alle allein mit dem Offset – eine ökonomische Sache.

Fassen wir die Eigenschaften von Segmenten und Offsets zusammen:

- Jedes Segment fasst maximal 64 kByte, weil die Register der CPU nur 16 Bit breit sind.
- Speicheradressen innerhalb eines Segments erreicht man mit Hilfe des Offsets.
- Es gibt 65.536 Segmente (2^{16}).
- Die Segmentanfänge liegen immer 16 Byte weit auseinander.
- Segmente können sich gegenseitig überlappen.
- Die Adressierung mit Segmenten und Offset erfolgt nur im *Real Mode* so wie beschrieben.

Neuere Prozessoren und Betriebssysteme besitzen die genannte Adressregister-Beschränkung nicht mehr. Ab dem 80386 können bis zu 4 GB linear adressiert werden. Deshalb setzen wir uns nicht weiter im Detail mit Segmenten und Offset auseinander. Die Programme jedoch, die wir im Folgenden entwickeln, sind dennoch höchstens 64 kByte groß und kommen deshalb mit einem Segment aus.

4 Das erste Assemblerprogramm

4.1 Die Befehle MOV und XCHG

Der `mov`-Befehl *(move)* ist wohl einer der am häufigsten verwendeten Befehle im Assembler. Er hat die folgende Syntax:

```
mov op1, op2
```

Mit dem `mov`-Befehl wird der zweite Operand in den ersten Operanden kopiert. Der erste Operand wird auch als Zieloperand, der zweite als Quelloperand bezeichnet. Beide Operanden müssen die gleiche Größe haben. Wenn der erste Operand beispielsweise die Größe von zwei Byte besitzt, muss auch der zweite Operand die Größe von zwei Byte besitzen.

Es ist nicht erlaubt, als Operanden das IP-Register zu benutzen. Wir werden später mit Sprungbefehlen noch eine indirekte Möglichkeit kennen lernen, dieses Register zu manipulieren.

Außerdem ist es nicht erlaubt, eine Speicherstelle in eine andere Speicherstelle zu kopieren. Diese Regel gilt für **alle** Assemblerbefehle mit zwei Operanden: Es dürfen niemals beide Operanden eine Speicherstelle ansprechen. Beim `mov`-Befehl hat dies zur Folge, dass, wenn eine Speicherstelle in eine zweite Speicherstelle kopiert werden soll, dies über ein Register erfolgen muss. Beispielsweise wird mit den zwei folgenden Befehlen der Inhalt von der Speicherstelle 0110h in die Speicherstelle 0112h kopiert:

```
mov ax, [0110]
mov [0112], ax
```

Der `xchg`-Befehl *(exchange)* hat die gleiche Syntax wie der `mov`-Befehl:

```
xchg  op1, op2
```

Wie sein Name bereits andeutet, vertauscht er den ersten und den zweiten Operanden. Die Operanden können allgemeine Register oder ein Register und eine Speicherstelle sein.

4.2 Das erste Programm

Schritt 1: Installieren des Assemblers

Laden Sie zunächst den Netwide-Assembler unter der folgenden Adresse herunter:

`http://sourceforge.net/project/showfiles.php?group_id=6208`

Dort sehen Sie eine Liste mit DOS 16-Bit-Binaries. Da wir unsere ersten Schritte unter DOS machen werden, laden Sie anschließend die Zip-Datei der aktuellsten Version herunter und entpacken diese.

Schritt 2: Assemblieren des ersten Programms

Anschließend kopieren Sie das folgende Programm in den Editor:

```
org 100h
start:
  mov ax, 5522h
  mov cx, 1234h
  xchg cx,ax
  mov al, 0
  mov ah,4Ch
  int 21h
```

Speichern Sie es unter der Bezeichnung „firstp.asm“. Anschließend starten Sie die Eingabeaufforderung "cmd.exe" und wechseln in das Verzeichnis, in das Sie den Assembler entpackt haben. Dort übersetzen Sie das Programm (wir nehmen an, dass der Quellcode im selben Verzeichnis steht wie der Assembler; ansonsten muss der Pfad natürlich mit angegeben werden):

```
nasm firstp.asm -f bin -o firstp.com
```

Mit der Option `-f` kann festgelegt werden, dass die übersetze Datei eine `*.COM` ist (dies ist die Voreinstellung des NASM). Bei diesem Dateityp besteht ein Programm aus nur einem Segment in dem sowohl Code, Daten wie auch der Stack liegt. Allerdings können COM-Programme auch nur maximal 64 KB groß werden. Eine komplette Liste aller möglichen Formate für `-f` können Sie sich mit der Option `-hf` ausgeben lassen. Über den Parameter `-o` legen Sie fest, dass Sie den Namen der assemblierten Datei selbst festlegen wollen.

Sie können das Programm nun ausführen, indem Sie `firstp` eingeben. Da nur einige Register hin und her geschoben werden, bekommen Sie allerdings nicht viel zu sehen.

Schritt 3: Analyse

Das Programm besteht sowohl aus Assembleranweisungen als auch Assemblerbefehlen. Nur die Assemblerbefehle werden übersetzt. Assembleranweisungen hingegen enthalten wichtige Informationen für den Assembler, die er beim Übersetzen des Programms benötigt.

Die Assembleranweisung ORG *(origin)* sagt dem Assembler, an welcher Stelle das Programm beginnt. Wir benötigen diese Assembleranweisung, weil Programme mit der Endung `.COM` immer mit Adresse 100h beginnen. Dies ist wichtig, wenn der Assembler eine Adresse beispielsweise für Sprungbefehle berechnen muss. In unserem sehr einfachen Beispiel muss der Assembler keine Adresse berechnen, weshalb wir die Anweisung in diesem Fall auch hätten weglassen können. Mit `start` wird schließlich der Beginn des Programmcodes festgelegt.

Für die Analyse des Programms benutzen wir den Debugger, der bereits seit MS-DOS mitgeliefert wurde. Er ist zwar nur sehr einfach und Textorientiert, hat jedoch den Vorteil, dass er nicht erst installiert werden muss und kostenlos ist.

Starten Sie den Debugger mit `debug firstp.com`. Wenn Sie den Dateinamen beim Starten des Debuggers nicht mit angeben haben, können sie dies jederzeit über `n firstp.com` (n steht für engl. *name*) und dem Befehl `l` *(load)* nachholen.

Anschließend geben Sie den Befehl `r` ein, um sich den Inhalt aller Register anzeigen zu lassen:

```
-r
AX=0000  BX=0000  CX=000C  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0CDC  ES=0CDC  SS=0CDC  CS=0CDC  IP=0100   NV UP EI PL NZ NA PO NC
0CDC:0100 B82255        MOV     AX,5522
```

In der ersten Zeile stehen die Datenregister sowie die Stackregister. Ihr Inhalt ist zufällig und kann sich von Aufruf zu Aufruf unterscheiden. Nur das BX-Register ist immer auf 0 gestellt und das CX-Register gibt die Größe des Programmcodes an. Sämtliche Angaben sind in hexadezimaler Schreibweise angegeben.

In der nächsten Zeile befinden sich die Segmentregister sowie der Befehlszähler. Die Register CS:IP zeigen auf den ersten Befehl in unserem Programm. Wie wir vorher gesehen haben können wir die tatsächliche Adresse über die Formel

Physikalische Adresse = Offsetadresse + Segmentadresse * 16_{dez}

errechnen. Setzen wir die Adresse für das Offset und das Segment ein, so erhalten wir

Physikalische Adresse = 0x0CDC * 16_{dez} + 0x0100 = 0xCDC0 + 0x0100 = 0xCEC0.

Dahinter befinden sich die Zustände der Flags. Vielleicht werden Sie sich wundern, dass ein Flag fehlt: Es ist das *Trap Flag,* das dafür sorgt, dass nach jeder Anweisung der Interrupt 1 ausgeführt wird, um es zu ermöglichen, dass das Programm Schritt für Schritt abgearbeitet wird. Es ist immer gesetzt, wenn ein Programm mit dem Debugger aufgeführt wird.

In der dritten und letzten Zeile sehen Sie zu Beginn nochmals die Adresse des Befehls, die immer mit dem Registerpaar CS:IP übereinstimmt. Dahinter befindet sich der Opcode des Befehls. Wir merken uns an dieser Stelle, dass der Opcode für den Befehl `MOV AX,5522` 3 Byte groß ist (zwei Hexadezimalziffern entsprechen immer einem Byte).

Mit der Anweisung `t` *(trace)* führen wir nun den ersten Befehl aus und erhalten die folgende Ausgabe:

```
-t
AX=5522  BX=0000  CX=000C  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0CDC  ES=0CDC  SS=0CDC  CS=0CDC  IP=0103   NV UP EI PL NZ NA PO NC
0CDC:0103 B93412        MOV     CX,1234
```

Wir haben die Register hervorgehoben, die sich verändert haben. Der Befehl `MOV` *(move)* hat dafür gesorgt, dass `5522` in das AX-Register kopiert wurde.

Der Prozessor hat außerdem den Inhalt des Programmzählers erhöht – und zwar um die Anzahl der Bytes, die der letzte Befehl im Arbeitsspeicher benötigte (also unsere 3 Bytes). Damit zeigt das Registerpaar CS:IP nun auf den nächsten Befehl.

Hat der Assembler nun wiederum den Befehl MOV CX,1234 abgearbeitet, erhöht er das IP-Register wiederum um die Größe des Opcodes und zeigt nun wiederum auf die Adresse

des nächsten Befehls:

```
-t
AX=5522  BX=0000  CX=1234  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0CDC  ES=0CDC  SS=0CDC  CS=0CDC  IP=0106  NV UP EI PL NZ NA PO NC
0CDC:0106 91            XCHG    CX,AX
```

Der `XCHG`-Befehl vertauscht nun die Register AX und CX, so dass jetzt AX den Inhalt von CX hat und CX den Inhalt von AX.

Wir haben nun genug gesehen und beenden deshalb das Programm an dieser Stelle. Dazu rufen wir über den Interrupt 21h die Betriebssystemfunktion 4Ch auf. Der Wert in AL (hier 0 für erfolgreiche Ausführung) wird dabei an das Betriebssystem zurückgegeben (er kann z. B. in Batch-Dateien über `%ERRORLEVEL%` abgefragt werden). Den Debugger können Sie nun beenden, indem Sie `q` (quit) eingeben.

4.3 „Hello World"-Programm

Es ist ja inzwischen fast üblich geworden, ein „Hello World"-Programm in einer Einführung in eine Sprache zu entwickeln. Dem wollen wir natürlich folgen:

```
org 100h
start:
 mov dx,hello_world
 mov ah,09h
 int 21h
 mov al, 0
 mov ah,4Ch
 int 21h
section .data
 hello_world: db 'hello, world', 13, 10, '$'
```

Nachdem Sie das Programm übersetzt und ausgeführt haben, starten Sie es mit dem Debugger:

```
-r
AX=0000  BX=0000  CX=001B  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0CDC  ES=0CDC  SS=0CDC  CS=0CDC  IP=0100   NV UP EI PL NZ NA PO NC
0CDC:0100 BA0C01        MOV     DX,010C
- q
```

Wie Sie erkennen können, hat der Assembler `hello_world` durch die Offsetadresse 010C ersetzt. Diese Adresse zusammen mit dem CS-Register entspricht der Stelle im Speicher, in der sich der Text befindet. Wir müssen diese der Betriebssystemfunktion 9h des Interrupts 21 übergeben, die dafür sorgt, dass der Text auf dem Bildschirm ausgegeben wird. Wie kommt der Assembler auf diese Adresse?

Die Anweisung „`section .data`" bewirkt, dass hinter dem letzten Befehl des Programms ein Bereich für Daten reserviert wird. Am Beginn dieses Datenbereichs wird der Text `'hello, world', 13, 10, '$'` bereitgestellt. Ab welcher Adresse der Datenbereich beginnt, lässt sich nicht genau vorhersagen. Wieso?

Wir können zunächst bestimmen, wo unser Programmcode beginnt und wie lang unser Programmcode ist.

COM-Programme beginnen immer erst bei der Adresse 100h (sonst kann sie das Betriebssystem nicht ausführen). Die Bytes von 0 bis FFh hat das Betriebssystem für Verwaltungszwecke reserviert. Dies muss natürlich dem Assembler mit der Anweisung `org 100h` mitgeteilt werden, der sonst davon ausgehen würde, dass das Programm bei Adresse 0 beginnt. Es sei hier nochmals darauf hingewiesen, dass Assembleranweisungen nicht mit übersetzt werden, sondern lediglich dazu dienen, dass der Assembler das Programm richtig übersetzen kann.

Der Opcode von `mov dx,010C` hat eine Größe von 3 Byte, der Opcode von `mov ah,09` und `mov ah, 4Ch` jeweils 2 Byte und der Opcode der Interruptaufrufe nochmals jeweils 2 Byte. Der Programmcode hat somit eine Größe von 11 Byte dezimal oder B Byte hexadezimal. Woher ich das weiß? Entweder kompiliere ich das Programm und drucke das Listing aus, oder ich benutze das DEBUG-Programm und tippe mal eben schnell die paar Befehle ein. Wie auch immer:

Das letzte Byte des Programms hat die Adresse 10Ah. Das ist auf den ersten Blick verblüffend, aber die Zählung hat ja nicht mit 101h, sondern mit 100h begonnen. Das erste freie Byte hinter dem Programmcode hat also die Adresse 10Bh, ab dieser Adresse könnten Daten stehen. Tatsächlich beginnt der Datenbereich erst an 10Ch, weil die meisten Compiler den Speicher ab geradzahligen oder durch vier teilbaren Adressen zuweisen.

Warum ist das so? Prozessoren lesen (und schreiben) 32 Bit gleichzeitig aus dem Speicher. Wenn eine 32-Bit-Variable an einer durch vier teilbaren Speicheradresse beginnt, kann sie von der CPU mit einem Speicherzugriff gelesen werden. Beginnt die Variable an einer anderen Speicheradresse, sind zwei Speicherzyklen notwendig.

Nun hängt es vom verwendeten Assemblerprogramm und dessen Voreinstellungen ab, welcher Speicherplatz dem Datensegment zugewiesen wird. Ein „kluges“ Assemblerprogramm wird eine 16-Bit-Variable oder eine 32-Bit-Variable stets so anordnen, dass sie mit einem Speicherzugriff gelesen werden kann. Nötigenfalls bleiben einige Speicherplätze ungenutzt. Wenn der Assembler allerdings die Voreinstellung hat, kein einziges Byte zu vergeuden, obwohl dadurch das Programm langsamer wird, tut er auch das. Die meisten Assembler werden den Datenbereich ab 10Ch oder 110h beginnen lassen.

Eine weitere Anweisung in diesem Programm ist die DB-Anweisung. Durch sie wird byteweise Speicherplatz reserviert. Der NASM kennt darüber hinaus noch weitere Anweisungen, um Speicherplatz zu reservieren. Dies sind:

Anweisung	**Breite**	**Bezeichnung**
DW	2 Byte	Word
DD	4 Byte	Doubleword
DF	6 Byte	-
DQ	8 Byte	Quadword
DT	10 Byte	Ten Byte

Die Zahl 13 dezimal entspricht im ASCII Zeichensatz dem Wagenrücklauf, die Zahl 10 dezimal entspricht im ASCII-Zeichensatz dem Zeilenvorschub. Beide zusammen setzen den Cursor auf den Anfang der nächsten Zeile. Das `$`-Zeichen wird für den Aufruf der Funktion 9h des Interrupts 21h benötigt und signalisiert das Ende der Zeichenkette.

4.4 EXE-Dateien

Die bisher verwendeten COM-Dateien können maximal ein Segment groß werden. EXE-Dateien können dagegen aus mehreren Segmenten bestehen und damit größer als 64 KB werden.

Im Gegensatz zu DOS unterstützt Windows überhaupt keine COM-Dateien mehr und erlaubt nur EXE-Dateien auszuführen. Trifft Windows auf eine COM-Datei, wird sie automatisch in der Eingabeaufforderung als MS-DOS-Programm ausgeführt.

Im Gegensatz zu einer COM-Datei besteht eine EXE-Datei nicht nur aus ausführbarem Code, sondern besitzt einen Header, der vom Betriebssystem ausgewertet wird. Der Header unterscheidet sich zwischen Windows und DOS. Wir wollen jedoch nicht näher auf die Unterschiede eingehen.

Ein weiterer Unterschied ist, dass EXE-Dateien in zwei Schritten erzeugt werden. Beim Übersetzen des Quellcodes entsteht zunächst eine „Objektdatei“ als Zwischenprodukt. Eine oder mehrere Objektdateien werden anschließend von einem Linker zum EXE-Programm zusammengefügt. Bei einem Teil der Objektdateien handelt es sich üblicherweise um Dateien aus Programmbibliotheken.

Der Netwide-Assembler besitzt allerdings selbst keinen Linker. Sie müssen deshalb entweder auf einen Linker eines anderen Compilers oder auf einen freien Linker wie beispielsweise ALINK zurückgreifen. ALINK kann von der Webseite `http://alink.sourceforge.net/` heruntergeladen werden.

Das folgende Programm ist eine Variante unseres „Hello World“-Programms.

```
segment code

start:
mov ax, data
mov ds, ax

mov dx, hello
mov ah, 09h
int 21h

mov al, 0
mov ah, 4Ch
int 21h

segment data
hello: db 'Hello World!', 13, 10, '$'
```

Speichern Sie das Programm unter `hworld.asm` ab und übersetzen Sie es:

```
nasm hworld.asm -fobj -o hworld.obj
alink hworld.obj
```

Beim Linken des Programms gibt der Linker möglicherweise eine Warnung aus. Bei *ALINK* lautet sie beispielsweise `Warning - no stack`. Beachten Sie diese Fehlermeldung nicht. Wir werden in einem der nächsten Abschnitte erfahren, was ein Stack ist.

Die erste Änderung, die auffällt, sind die Assembleranweisungen `segment code` und `segment data`. Die Anweisung ist ein Synonym für `section`. Wir haben uns hier für `segment` entschieden, damit deutlicher wird, dass es sich hier im Unterschied zu einer COM-Datei tatsächlich um verschiedene Segmente handelt.

Die Bezeichnungen für die Segmente sind übrigens beliebig. Sie können die Segmente auch `hase` und `igel` nennen. Der Assembler benötigt die Bezeichnung lediglich für die Adressberechnung und muss dazu wissen, wo ein neues Segment anfängt. Allerdings ist es üblich das Codesegment mit `code` und das Datensegment mit `data` zu bezeichnen.

Die Anweisung `start:` legt den Anfangspunkt des Programms fest. Diese Information benötigt der Linker, wenn er mehrere Objektdateien zusammenlinken muss. In diesem Fall muss er wissen, in welcher Datei sich der Eintrittspunkt befindet.

In einer Intel-CPU gibt es keinen direkten Befehl, eine Konstante in ein Segmentregister zu laden. Man muss deshalb entweder den Umweg über ein universelles Register gehen (im Beispiel: AX), oder man benutzt den Stack[1] (den man vorher korrekt initialisieren muss): `push data`, dann `pop data`.

Nachdem Sie das Programm übersetzt haben, führen Sie es mit dem Debugger aus:

```
debug hworld.exe
-r
AX=0000  BX=FFFF  CX=FE6F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0D3A  ES=0D3A  SS=0D4A  CS=0D4A  IP=0000   NV UP EI PL NZ NA PO NC
0D4A:0000 B84B0D        MOV     AX,0D4C
-t

AX=0D4C  BX=FFFF  CX=FE6F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0D3A  ES=0D3A  SS=0D4A  CS=0D4A  IP=0003   NV UP EI PL NZ NA PO NC
0D4A:0003 8ED8          MOV     DS,AX
-t

AX=0D4B  BX=FFFF  CX=FE6F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0D4C  ES=0D3A  SS=0D4A  CS=0D4A  IP=0005   NV UP EI PL NZ NA PO NC
0D4A:0005 BA0000        MOV     DX,0000
```

Die ersten zwei Zeilen unterscheiden sich vom COM-Programm. Der Grund hierfür ist, dass wir nun zunächst das Datensegmentregister (DS) initialisieren müssen. Bei einem COM-Programm sorgt das Betriebssystem dafür, dass alle Segmentregister einen Anfangswert haben, und zwar alle den gleichen Wert. Bei einem COM-Programm braucht man sich deshalb nicht um die Segmentregister kümmern, hat dafür aber auch nur 64k für Programm + Daten zur Verfügung. Bei einem EXE-Programm entfällt die Einschränkung auf 64 k, aber man muss sich selbst um die Segmentregister kümmern.

Wie Sie sehen, beträgt die Differenz zwischen DS- und CS-Register anschließend 2 (0D4C – 0D4A). Das bedeutet allerdings nicht, dass nur zwei Byte zwischen Daten- und Codesegment liegt - darin würde der Code auch keinen Platz finden. Wie wir im letzten Abschnitt gesehen haben, liegen die Segmente 16 Byte auseinander, weshalb DS und CS Segment tatsächlich 32 Byte auseinanderliegen.

Der DX-Register erhält diesmal den Wert 0, da die Daten durch die Trennung von Daten- und Codesegment nun ganz am Anfang des Segments liegen.

1 `http://de.wikibooks.org/wiki/Stack`

Der Rest des Programms entspricht dem einer COM-Datei. Beenden Sie deshalb anschließend den Debugger.

4.5 Hello World in anderen Betriebssystemen

Die oben angegebenen Programme enthalten zwei verschiedene Systemabhängigkeiten: Zum einen logischerweise die Abhängigkeit vom Prozessor (es handelt sich um Code für den 80x86, der logischerweise nicht auf einem anderen Prozessor lauffähig ist), aber zusätzlich die Abhängigkeit vom verwendeten Betriebssystem (in diesem Fall DOS; beispielsweise wird obiger Code unter Linux nicht laufen). Um die Unterschiede zu zeigen, sollen hier exemplarisch Versionen von Hello World für andere Betriebssysteme gezeigt werden. Der verwendete Assembler ist in allen Fällen nasm.

Eine Anmerkung vorweg: Die meisten Betriebssysteme verwenden den seit dem 80386 verfügbaren 32-Bit-Modus. Dieser wird später im Detail behandelt, im Moment ist nur interessant, dass die Register 32 Bits haben und ein „e“ (für *extended* = erweitert) am Anfang des Registernamens bekommen (z. B. `eax` statt `ax`), und dass man Segmentregister in der Regel vergessen kann.

4.5.1 Linux

Unter Linux werden statt Segmenten sections verwendet (letztlich landet alles im selben 32-Bit-Segment). Die Code-Section heißt „`.text`“, die Datensection „`.data`“ und der Einsprungpunkt für den Code „`_start`“. Systemaufrufe benutzen int 80h statt des 21h von DOS, Werte werden in der Regel wie unter DOS über Register übergeben (wenngleich die Einzelheiten sich unterscheiden). Außerdem wird ein Zeilenende unter Linux nur mit einem Linefeed-Zeichen (ASCII 10) gekennzeichnet. Es sollte nicht schwer sein, die einzelnen Elemente des Hello-World-Programms im folgenden Quelltext wiederzuerkennen.

```
section .text
global _start
_start:
    mov ecx, hello
    mov edx, length
    mov ebx, 1      ; Dateinummer der Standard-Ausgabe
    mov eax, 4      ; Funktionsnummer: Ausgabe
    int 80h
    mov ebx, 0
    mov eax, 1      ; Funktionsnummer: Programm beenden
    int 80h

section .data
    hello  db 'Hello World!', 10
    length equ $ - hello;
```

Die folgenden Kommandozeilenbefehle bewirken das Kompilieren und Ausführen des Programms:

```
nasm -g -f elf32 hello.asm
ld hello.o -o hello
./hello
```

Erläuterung: `nasm` ist wieder der Assembler. Er erzeugt hier die Datei `hello.o`, die man noch nicht ausführen kann. Die Option `-g` sorgt dafür, dass Debuginformationen in die zu erzeugende `hello.o`-Datei geschrieben werden. Der sogenannte Linker `ld` erzeugt aus `hello.o` die fertige, lauffähige Datei `hello`. Der letzte Befehl startet schließlich das Programm.

Hat man den GNU Debugger[2] am Start, kann man per

```
gdb hello
```

ähnlich wie unter *MS Windows* debuggen. Der GDB öffnet eine Art Shell, in der diverse Befehle verfügbar sind. Der Befehl `list` gibt den von uns verfassten Assemblercode inklusive Zeilennummern zurück. Mit `break 5` setzen wir einen Breakpoint in der fünften Zeile unseres Codes. Ein anschließendes `run` führt das Programm bis zum Breakpoint aus, und wir können nun mit `info registers` die Register auslesen. Einen einzelnen Schritt können wir mit `stepi` ausführen.

```
(gdb) list
1       section .text
2       global _start
3       _start:
4           mov ecx, hello
5           mov edx, length
6           mov ebx, 1      ; Dateinummer der Standard-Ausgabe
7           mov eax, 4      ; Funktionsnummer: Ausgabe
8           int 80h
9           mov ebx, 0
10          mov eax, 1      ; Funktionsnummer: Programm beenden
(gdb) break 5
Breakpoint 1 at 0x8048085: file hello.asm, line 5.
(gdb) run
Starting program: /home/name/projects/assembler/hello_world/hello

Breakpoint 1, _start () at hello.asm:5
5           mov edx, length
(gdb) info registers
eax            0x0      0
ecx            0x80490a4        134516900
edx            0x0      0
ebx            0x0      0
esp            0xbf9e6140       0xbf9e6140
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048085        0x8048085 <_start+5>
eflags         0x200292 [ AF SF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
```

2 http://de.wikipedia.org/wiki/GNU_Debugger

```
gs             0x0      0
(gdb) stepi
_start () at hello.asm:6
6           mov ebx, 1      ; Dateinummer der Standard-Ausgabe
(gdb) quit
```

4.5.2 NetBSD

Die NetBSD-Version unterscheidet sich von der Linux-Version vor allem dadurch, dass alle Argumente über den Stack übergeben werden. Auch der Stack wird später behandelt; an dieser Stelle ist nur wesentlich, dass man mit `push`-Werte auf den Stack bekommt, und dass die Reihenfolge wesentlich ist. Außerdem benötigt man für NetBSD eine spezielle `.note`-Section, die das Programm als NetBSD-Programm kennzeichnet.

```
section .note.netbsd.ident
    dd 0x07,0x04,0x01
    db "NetBSD",0x00,0x00
    dd 200000000

section .text
global _start
_start:
    push dword length
    push dword hello
    push dword 1
    mov        eax, 4
    push eax
    int        80h
    push dword 0
    mov eax, 1
    push eax
    int 80h

section .data
    hello  db 'Hello World!', 10
    length equ $ - hello;
```

5 Rechnen mit dem Assembler

5.1 Die Addition

Für die Addition stehen zwei Befehle zu Verfügung: `add` und `adc`. Der `adc` *(Add with Carry)* berücksichtigt das *Carry Flag*. Wir werden weiter unten noch genauer auf die Unterschiede eingehen.

Die Syntax von `add` und `adc` ist identisch:

```
add/adc Zieloperand, Quelloperand
```

Beide Befehle addieren den Quell- und Zieloperanden und speichern das Resultat im Zieloperanden ab. Ziel- und Quelloperand können entweder ein Register oder eine Speicherstelle sein (natürlich darf nur entweder der Ziel- oder der Quelloperand eine Speicherstelle sein, niemals aber beide zugleich).

Das folgende Programm addiert zwei Zahlen miteinander und speichert das Ergebnis in einer Speicherstelle ab:

```
org 100h
start:
  mov bx, 500h
  add bx, [summand1]
  mov [ergebnis], bx
  mov ah, 4Ch
  int 21h
section .data
  summand1 DW 900h
  ergebnis DW 0h
```

Unter Linux gibt es dabei - bis auf die differierenden Interrupts und die erweiterten Register - kaum Unterschiede:

```
section .text
global _start
_start:
  mov ebx,500h
  add ebx,[summand1]
  mov [ergebnis],ebx
  ; Programm ordnungsgemäß beenden
  mov eax,1
  mov ebx,0
  int 80h
section .data
  summand1 dd 900h
  ergebnis dd 0h
```

Es fällt auf, dass `summand1` und `ergebnis` in eckigen Klammern geschrieben sind. Der Grund hierfür ist, dass wir nicht die Adresse benötigen, sondern den Inhalt der Speicherzelle.

Fehlen die eckigen Klammern, interpretiert der Assembler das Label `summand1` und `ergebnis` als Adresse. Im Falle des `add`-Befehls würde das BX Register folglich mit der Adresse von `summand1` addiert. Beim `mov`-Befehl hingegen würde dies der Assembler mit einer Fehlermeldung quittieren, da es nicht möglich ist, den Inhalt des BX Registers in eine Adresse zu kopieren und es folglich keinen Opcode gibt.

Wir verfolgen nun wieder mit Hilfe des Debuggers die Arbeit unseres Programms:

```
-r
AX=0000  BX=0000  CX=0014  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0CDC  ES=0CDC  SS=0CDC  CS=0CDC  IP=0100   NV UP EI PL NZ NA PO NC
0CDC:0100 BB0005        MOV     BX,0500
-t

AX=0000  BX=0500  CX=0014  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0CDC  ES=0CDC  SS=0CDC  CS=0CDC  IP=0103   NV UP EI PL NZ NA PO NC
0CDC:0103 031E1001      ADD     BX,[0110]                           DS:0110=0900
```

Wir sehen an DS:0110=0900, dass ein Zugriff auf die Speicherstelle 0110 im Datensegment erfolgt ist. Wie wir außerdem erkennen können, ist der Inhalt der Speicherzelle 0900.

Abschließend speichern wir das Ergebnis unserer Berechnung wieder in den Arbeitsspeicher zurück:

```
AX=0000  BX=0E00  CX=0014  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0CDC  ES=0CDC  SS=0CDC  CS=0CDC  IP=0107   NV UP EI PL NZ NA PE NC
0CDC:0107 891E1201      MOV     [0112],BX                           DS:0112=0000
-t
```

Wir lassen uns nun die Speicherstelle 0112, in der das Ergebnis gespeichert ist, über den `d`-Befehl ausgeben:

```
-d ds:0112 L2
0CDC:0110        00 0E
```

Die Ausgabe überrascht ein wenig, denn der Inhalt der Speicherstelle 0112 unterscheidet sich vom (richtigen) Ergebnis 0E00 im BX Register. Der Grund hierfür ist eine Eigenart der 80x86 CPU: Es werden *High* und *Low Byte* vertauscht. Als *High Byte* bezeichnet man die höherwertige Hälfte, als *Low Byte* die niederwertige Hälfte eines 16 Bit Wortes. Um das richtige Ergebnis zu erhalten, muss entsprechend wieder *Low* und High Byte*vertauscht werden.*

Unter Linux sieht der Debugging-Ablauf folgendermaßen aus:

```
(gdb) break 6
Breakpoint 1 at 0x8048085: file add.asm, line 6.
(gdb) run
Starting program: /(...)/(...)/add

Breakpoint 1, _start () at add.asm:6
6       add ebx,[summand1]
(gdb) info registers
eax            0x0    0
ecx            0x0    0
edx            0x0    0
```

```
ebx            0x500    1280
esp            0xbffe0df0    0xbffe0df0
ebp            0x0    0x0
esi            0x0    0
edi            0x0    0
eip            0x8048085    0x8048085 <_start+5>
eflags         0x200392    [ AF SF TF IF ID ]
cs             0x73    115
ss             0x7b    123
ds             0x7b    123
es             0x7b    123
fs             0x0    0
gs             0x0    0
(gdb) stepi
_start () at add.asm:7
7        mov [ergebnis],ebx
(gdb) info registers
eax            0x0    0
ecx            0x0    0
edx            0x0    0
ebx            0xe00    3584
esp            0xbffe0df0    0xbffe0df0
ebp            0x0    0x0
esi            0x0    0
edi            0x0    0
eip            0x804808b    0x804808b <_start+11>
eflags         0x200306    [ PF TF IF ID ]
cs             0x73    115
ss             0x7b    123
ds             0x7b    123
es             0x7b    123
fs             0x0    0
gs             0x0    0
```

Wie Sie sehen, lässt sich der GDB etwas anders bedienen. Nachdem wir einen Breakpoint auf die Zeile 6 gesetzt haben (Achtung: Das bedeutet, dass die sechste Zeile nicht mehr mit ausgeführt wird!), führen wir das Programm aus und lassen uns den Inhalt der Register ausgeben. Das `ebx`-Register enthält die Zahl 500h (dezimal 1280), die wir mit dem `mov`-Befehl hineingeschoben haben. Mit dem GDB-Kommando `stepi` rücken wir in die nächste Zeile (die den `add`-Befehl enthält). Erneut lassen wir uns die Register auflisten. Das `ebx`-Register enthält nun die Summe der Addition: 0xe00 (dezimal 3584).

Im Folgenden wollen wir eine 32-Bit-Zahl addieren. Bei einer 32-Bit-Zahl vergrößert sich der darstellbare Bereich für vorzeichenlose Zahlen von 0 bis 65.535 auf 0 bis 4.294.967.295 und für vorzeichenbehafteten Zahlen von –32.768 bis +32.767 auf –2.147.483.648 bis +2.147.483.647. Die einfachste Möglichkeit bestünde darin, ein 32-Bit-Register zu benutzen, was ab der 80386-CPU problemlos möglich ist. Wir wollen hier allerdings die Verwendung des `adc`-Befehls zeigen, weshalb wir davon nicht Gebrauch machen werden.

Für unser Beispiel benutzen wir die Zahlen 188.866 und 103.644 (Dezimal). Wir schauen uns die Rechnung im Dualsystem an:

```
    10 11100001 11000010 (188.866)
+    1 10010100 11011100 (103.644)
Ü   11        11 1
---------------------------------
  100 01110110 10011110 (292.510)
```

Wie man an der Rechnung erkennt, findet vom 15ten nach dem 16ten Bit ein Übertrag statt (fett hervorgehoben). Der Additionsbefehl, der die oberen 16 Bit addiert, muss dies berücksichtigen.

Die Frage, die damit aufgeworfen wird ist, wie wird der zweite Additionsbefehl davon in Kenntnis gesetzt, dass ein Übertrag stattgefunden hat oder nicht. Die Antwort lautet: Der erste Additionsbefehl, der die unteren 16 Bit addiert, setzt das *Carry Flag,* wenn ein Übertrag stattfindet, andernfalls löscht er es. Der zweite Additionsbefehl, der die oberen 16 Bit addiert, muss nun eins hinzuaddieren, wenn das *Carry Flag* gesetzt ist. Genau dies tut der `adc`-Befehl (im Gegensatz zum `add`-Befehl) auch.

Das folgende Programm addiert zwei 32-Bit-Zahlen miteinander:

```
org 100h
start:
  mov ax, [summand1]
  add ax, [summand2]
  mov [ergebnis], ax

  mov ax, [summand1+2]
  adc ax, [summand2+2]
  mov [ergebnis+2], ax

  mov ah, 4Ch
  int 21h
section .data
  summand1 DD 2E1C2h
  summand2 DD 194DCh
  ergebnis DD 0h
```

Die ersten drei Befehle entsprechen unserem ersten Programm. Dort werden Bit 0 bis 15 addiert. Der `add`-Befehl setzt außerdem das *Carry Flag,* wenn es einen Übertrag von der 15ten auf die 16te Stelle gab (was hier der Fall ist, wie wir auch gleich mit Hilfe des Debuggers nachprüfen werden).

Mit den nächsten drei Befehlen werden Bit 16 bis 31 addiert. Deshalb müssen wir dort zwei Byte zur Adresse hinzuaddieren und außerdem mit dem `adc`-Befehl das *Carry Flag* berücksichtigen.

Wir sehen uns das Programm wieder mit dem Debugger an. Dabei sehen wir, dass der `add`-Befehl das *Carry Flag* setzt:

```
AX=E1C2  BX=0000  CX=0024  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0CDC  ES=0CDC  SS=0CDC  CS=0CDC  IP=0103   NV UP EI PL NZ NA PO NC
0CDC:0103 03061C01      ADD     AX,[011C]                          DS:011C=94DC
-t

AX=769E  BX=0000  CX=0024  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0CDC  ES=0CDC  SS=0CDC  CS=0CDC  IP=0107   OV UP EI PL NZ NA PO CY
0CDC:0107 A32001        MOV     [0120],AX                          DS:0120=0000
```

Mit NC zeigt der Debugger an, dass das *Carry Flag* nicht gesetzt ist, mit CY, dass es gesetzt ist. Das *Carry Flag* ist allerdings nicht das einzige Flag, das bei der Addition beeinflusst wird:

- Das *Zero Flag* ist gesetzt, wenn das Ergebnis 0 ist.

- Das *Sign Flag* ist gesetzt, wenn das *Most Significant Bit* den Wert 1 hat. Dies kann auch der Fall sein, wenn nicht mit vorzeichenbehafteten Zahlen gerechnet wird. In diesem Fall kann das *Sign Flag* ignoriert werden.
- Das *Parity Bit* ist gesetzt, wenn das Ergebnis eine gerade Anzahl von Bits erhält. Es dient dazu, Übertragungsfehler festzustellen. Wir gehen hier aber nicht näher darauf ein, da es nur noch selten benutzt wird.
- Das *Auxiliary Carry Flag* entspricht dem *Carry Flag,* wird allerdings benutzt, wenn mit BCD-Zahlen gerechnet werden soll.
- Das *Overflow Flag* wird gesetzt, wenn eine negative Zahl nicht mehr darstellbar ist, weil das Ergebnis zu groß geworden ist.

5.2 Subtraktion

Die Syntax von `sub` *(subtract)* und `sbb` *(subtract with borrow)* ist äquivalent mit dem `add/adc`-Befehl:

```
sub/sbb Zieloperand, Quelloperand
```

Bei der Subtraktion muss die Reihenfolge von Ziel- und Quelloperand beachtet werden, da die Subtraktion im Gegensatz zur Addition nicht kommutativ ist. Der `sub/sbb`-Befehl zieht vom Zieloperanden den Quelloperanden ab (Ziel = Ziel – Quelle):

Beispiel: 70 – 50 = 20

Diese Subtraktion kann durch die folgenden zwei Anweisungen in Assembler dargestellt werden:

```
mov ax,70h
sub ax,50h
```

Wie bei der Addition können sich auch bei der Subtraktion die Operanden in zwei oder mehr Registern befinden. Auch hier wird das *Carry Flag* verwendet. So verwundert es nicht, dass die Subtraktion einer 32-Bit-Zahl bei Verwendung von 16-Bit-Register fast genauso aussieht wie das entsprechende Additionsprogramm:

```
org 100h
start:
  mov ax, [zahl1]
  sub ax, [zahl2]
  mov [ergebnis], ax

  mov ax, [zahl1+2]
  sbb ax, [zahl2+2]
  mov [ergebnis+2], ax

  mov ah, 4Ch
  int 21h
section .data
  zahl1 DD 70000h
  zahl2 DD 50000h
  ergebnis DD 0h
```

Der einzige Unterschied zum entsprechenden Additionsprogramm besteht tatsächlich darin, dass anstelle des `add`-Befehls der `sub`-Befehl und anstelle des `adc`- der `sbb`-Befehl verwendet wurde.

Wie beim `add`- und `adc`-Befehl werden die Flags *Zero, Sign, Parity, Auxiliary Carry* und *Carry* gesetzt.

5.3 Setzen und Löschen des Carryflags

Nicht nur die CPU sondern auch der Programmierer kann das *Carry Flag* beeinflussen. Dazu existieren die folgenden drei Befehle:

- `stc` *(Set Carry Flag)* – setzt das *Carry Flag*
- `clc` *(Clear Carry Flag)* – löscht das *Carry Flag*
- `cmc` *(Complement Carry Flag)* – dreht den Zustand des *Carry Flag* um

5.4 Die Befehle INC und DEC

Der Befehl `inc` erhöht den Operanden um eins, der Befehl `dec` verringert den Operanden um eins. Die Befehle haben die folgende Syntax:

```
inc Operand
dec Operand
```

Der Operand kann entweder eine Speicherstelle oder ein Register sein. Beispielsweise wird über den Befehl

```
dec ax
```

der Inhalt des AX Registers um eins verringert. Der Befehl bewirkt damit im Grunde das Gleiche wie der Befehl `sub ax, 1`. Es gibt lediglich einen Unterschied: `inc` und `dec` beeinflussen nicht das *Carry Flag*.

5.5 Zweierkomplement bilden

Wir haben bereits mit dem Zweierkomplement gerechnet. Doch wie wird dieses zur Laufzeit gebildet? Die Intel-CPU hält einen speziellen Befehl dafür bereit, den `neg`-Befehl. Er hat die folgende Syntax:

```
neg Operand
```

Der Operand kann entweder eine Speicherzelle oder ein allgemeines Register sein. Um das Zweierkomplement zu erhalten, zieht der `neg`-Befehl den Operanden von 0 ab. Entsprechend wird das Carryflag gesetzt, wenn der Operand nicht null ist.

5.6 Die Multiplikation

Die Befehle `mul` *(**Mul**tiply unsigned)*und `imul` *(**I**nteger **Mul**tiply)* sind für die Multiplikation zweier Zahlen zuständig. Mit `mul` werden vorzeichenlose Ganzzahlen multipliziert, wohingegen mit `imul` vorzeichenbehaftete Ganzzahlen multipliziert werden. Der `mul`-Befehl besitzt nur einen Operanden:

```
mul Quelloperand
```

Der Zieloperand ist sowohl beim `mul`- wie beim `imul`-Befehl immer das AL- oder AX-Register. Der Quelloperand kann entweder ein allgemeines Register oder eine Speicherstelle sein.

Da bei einer 8-Bit-Multiplikation meistens eine 16-Bit-Zahl das Ergebnis ist (z. B.: 20 * 30 = 600) und bei einer 16-Bit-Multiplikation meistens ein 32-Bit-Ergebnis entsteht, wird das Ergebnis bei einer 8-Bit-Multiplikation immer im AX-Register gespeichert, bei einer 16-Bit-Multiplikation in den Registern DX:AX. Der höherwertige Teil wird dabei vom DX-Register aufgenommen, der niederwertige Teil vom AX-Register.

Beispiel:

```
org 100h
start:
  mov ax, 350h
  mul word[zahl]
  mov ah, 4Ch
  int 21h
section .data
  zahl     DW 750h
  ergebnis DD 0h
```

Wie Sie sehen, besitzt der Zeiger `zahl` das Präfix `word`. Dies benötigt der Assembler, da sich der Opcode für den `mul`-Befehl unterscheidet, je nachdem, ob es sich beim Quelloperand um einen 8-Bit- oder einen 16-Bit-Operand handelt. Wenn der Zieloperand 8 Bit groß ist, dann muss der Assembler den Befehl in Opcode F6h übersetzen, ist der Zieloperand dagegen 16 Bit groß, muss der Assembler den Befehl in den Opcode F7h übersetzen.

Ist der höherwertige Anteil des Ergebnisses 0, so werden *Carry Flag* und *Overflow Flag* gelöscht, ansonsten werden sie auf 1 gesetzt. Das *Sign Flag, Zero Flag, Auxilary Flag* und *Parity Flag* werden nicht verändert.

In der Literatur wird erstaunlicherweise oft „vergessen“, dass der `imul`-Befehl im Gegensatz zum `mul`-Befehl in drei Varianten existiert. Vielleicht liegt dies daran, dass diese Erweiterung erst mit der 80186-CPU eingeführt wurde (und stellt daher eine der wenigen Neuerungen der 80186-CPU dar). Aber wer programmiert heute noch für den 8086/8088?

Die erste Variante des `imul`-Befehls entspricht der Syntax des `mul`-Befehls:

```
imul Quelloperand
```

Eine Speicherstelle oder ein allgemeines Register wird entweder mit dem AX-Register oder dem Registerpaar DX:AX multipliziert.

Die zweite Variante des `imul`-Befehls besitzt zwei Operanden und hat die folgende Syntax:

```
imul Zieloperand, Quelloperand
```

Der Zieloperand wird mit dem Quelloperand multipliziert und das Ergebnis im Zieloperanden abgelegt. Der Zieloperand muss ein allgemeines Register sein, der Quelloperand kann entweder ein Wert, ein allgemeines Register oder eine Speicherstelle sein.

Die dritte Variante des `imul`-Befehls besitzt drei(!) Operanden. Damit widerlegt der Befehl die häufig geäußerte Aussage, dass die Befehle der Intel-80x86-Plattform entweder keinen, einen oder zwei Operanden besitzen können:

```
imul Zieloperand, Quelloperand1, Quelloperand2
```

Bei dieser Variante wird der erste Quelloperand mit dem zweiten Quelloperanden multipliziert und das Ergebnis im Zieloperanden gespeichert. Der Zieloperand und der erste Quelloperand müssen entweder ein allgemeines Register oder eine Speicherstelle sein, der zweite Quelloperand dagegen muss ein Wert sein.

Bitte beachten Sie, dass der DOS-Debugger nichts mit dem Opcode der zweiten und dritten Variante anfangen kann, da er nur Befehle des 8086/88er-Prozessors versteht. Weiterhin sollten Sie beachten, dass diese Varianten nur für den `imul`-Befehl existieren – also auch nicht für den `idiv`-Befehl.

5.7 Die Division

Die Befehle `div` *(Unsigned **Div**ide)* und `idiv` *(**I**nteger **Div**ision)* sind für die Division zuständig. Die Syntax der beiden Befehle entspricht der von `mul` und `imul`:

```
div  Quelloperand
idiv Quelloperand
```

Der Zieloperand wird dabei durch den Quelloperand geteilt. Der Quelloperand muss entweder eine Speicherstelle oder ein allgemeines Register sein. Der Zieloperand befindet sich immer im AX-Register oder im DX:AX-Register. In diesen Registern wird auch das Ergebnis gespeichert: Bei einer 8-Bit-Division befindet sich das Ergebnis im AL-Register und der Rest im AH-Register, bei der 16-Bit-Division befindet sich das Ergebnis im AX-Register und der Rest im DX-Register. Da die Division nicht kommutativ ist, dürfen die Operanden nicht vertauscht werden.

Beispiel:

```
org 100h
 start:
   mov dx,0010h
   mov ax,0A00h
   mov cx,0100h
   div cx          ; DX:AX / CX
   mov ah, 4Ch
   int 21h
```

Das Programm dürfte selbsterklärend sein. Es teilt das DX:AX-Register durch das CX-Register und legt das Ergebnis im AX-Register ab.

Da das Ergebnis nur im 16 Bit breiten AX-Register gespeichert wird, kann es passieren, dass ein Überlauf stattfindet, weil das Ergebnis nicht mehr in das Register passt. Einen Überlauf erzeugt beispielsweise der folgende Programmausschnitt:

```
mov dx, 5000h
mov ax, 0h
mov cx, 2h
div cx
```

Das Ergebnis ist größer als 65535. Wenn die CPU auf einen solchen Überlauf stößt, löst sie eine *Divide Error Exception* aus. Dies führt dazu, dass der Interrupt 0 aufgerufen und eine Routine des Betriebssystems ausgeführt wird. Diese gibt die Fehlermeldung „Überlauf bei Division“ aus und beendet das Programm. Auch die Divison durch 0 führt zum Aufruf der *Divide Error Exception.*

Wenn der Prozessor bei einem Divisionsüberlauf und bei einer Division durch 0 eine Exception auslöst, welche Bedeutung haben dann die Statusflags? Die Antwort lautet, dass sie bei der Division tatsächlich keine Rolle spielen, da sie keinen definierten Zustand annehmen.

5.8 Logische Operationen

In Allgemeinbildungsquiz findet man manchmal die folgende Frage: „Was ist die kleinste adressierbare Einheit eines Computers?“ Überlegen Sie einen Augenblick. Wenn Sie nun mit Bit geantwortet haben, so liegen Sie leider falsch. Der Assembler bietet tatsächlich keine Möglichkeit, ein einzelnes Bit im Arbeitsspeicher zu manipulieren. Die richtige Antwort ist deshalb, dass ein Byte die kleinste adressierbare Einheit eines Computers ist. Dies gilt im Großen und Ganzen auch für die Register. Nur das Flag-Register bildet hier eine Ausnahme, da hier einzelne Bits mit Befehlen wie `sti` oder `clc` gesetzt und zurückgesetzt werden können.

Um dennoch einzelne Bits anzusprechen, muss der Programmierer deshalb den Umweg über logische Operationen wie `AND, OR, XOR` und `NOT` gehen. Wie die meisten höheren Programmiersprachen sind auch dem Assembler diese Befehle bekannt.

Die nachfolgende Tabelle zeigt nochmals die Wahrheitstabelle der logischen Grundverknüpfungen:

A	B	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Ein kleines Beispiel zur Verdeutlichung:

```
    10110011
AND 01010001
------------
    00010001
```

Die logischen Verknüpfungen haben die folgende Syntax:

```
and Zieloperand, Quelloperand
or  Zieloperand, Quelloperand
xor Zieloperand, Quelloperand
```

Das Ergebnis der Verknüpfung wird jeweils im Zieloperanden gespeichert. Der Quelloperand kann ein Register, eine Speicherstelle oder ein Wert sein, der Zieloperand kann ein Register oder eine Speicherstelle sein (wie immer dürfen nicht sowohl Ziel- wie auch Quelloperand eine Speicherstelle sein).

Das *Carry* und *Overflow Flag* werden gelöscht, das Vorzeichen *Flag,* das *Null Flag* und das *Parity Flag* werden in Abhängigkeit des Ergebnisses gesetzt. Das *Auxiliary Flag* hat einen undefinierten Zustand.

Die `NOT`-Verknüpfung dreht den Wahrheitswert der Bits einfach um. Dies entspricht dem Einerkomplement einer Zahl. Der `not`-Befehl hat deshalb die folgende Syntax:

```
not Zieloperand
```

Der Zieloperand kann eine Speicherstelle oder ein Register sein. Die Flags werden nicht verändert.

In vielen Assemblerprogrammen sieht man übrigens häufig eine Zeile wie die folgende:

```
xor ax, ax
```

Da hier Quell- und Zielregister identisch sind, können die Bits nur entweder den Wert 0, 0 oder 1, 1 besitzen. Wie aus der Wahrheitstabelle ersichtlich, ist das Ergebnis in beiden Fällen 0. Der Befehl entspricht damit

```
mov ax, 0
```

Der `mov`-Befehl ist drei Byte lang, während die Variante mit dem `xor`-Befehl nur zwei Byte Befehlscode benötigt.

5.9 Schiebebefehle

Schiebebefehle verschieben den Inhalt eines Registers oder einer Speicherstelle bitweise. Mit den Schiebebefehlen lässt sich eine Zahl mit 2^n multiplizieren bzw. dividieren. Dies geschieht allerdings wesentlich schneller und damit effizienter als mit den Befehlen `mul` und `div`.

Der 8086 kennt vier verschiedene Schiebeoperationen: Links- und Rechtsverschiebungen sowie arithmetische und logische Schiebeoperationen. Bei arithmetischen Schiebeoperationen wird das Vorzeichen mit berücksichtigt:

- `sal` *(Shift Arithmetic Left):* arithmetische Linksverschiebung
- `sar` *(Shift Arithmetic Right):* arithmetische Rechtsverschiebung
- `shl` *(Shift Logical Left):* logische Linksverschiebung
- `shr` *(Shift Logical Right):* logische Rechtsverschiebung

Die Schiebebefehle haben immer die folgende Syntax:

```
sal / sar / shl / shr Zieloperand, Zähloperand
```

Der Zieloperand gibt an, welcher Wert geschoben werden soll. Der Zähloperand gibt an, wie oft geschoben werden soll. Der Zieloperand kann eine Speicherstelle oder ein Register sein. Der Zähloperand kann ein Wert oder das CL-Register sein.

Da der Zähloperand 8 Bit groß ist, kann er damit Werte zwischen 0 und 255 annehmen. Da ein Register allerdings maximal 32 Bit breit sein kann, sind nur Werte von 1 bis 31 sinnvoll. Alles was darüber ist, würde bedeuten, dass sämtliche Bits an den Enden hinausgeschoben werden. Der 8086-CPU war dies noch egal: Sie verschob auch Werte die größer als 31 sind, was allerdings entsprechend lange Ausführungszeiten nach sich ziehen konnte. Ab der 80286 werden deshalb nur noch die ersten 5 Bit beachtet, so dass maximal 31 Verschiebungen durchgeführt werden.

Das zuletzt herausgeschobene Bit geht zunächst einmal nicht verloren: Vielmehr wird es zunächst als *Carry Flag* gespeichert. Bei der logischen Verschiebung wird eine 0 nachgeschoben, bei der arithmetischen Verschiebung das *Most Significant Bit.*

Dies erscheint zunächst nicht schlüssig, daher wollen wir und die arithmetische und logischen Verschiebung an der Zahl –10 klar anschauen. Wir wandeln die Zahl zunächst in die Binärdarstellung um und bilden dann das Zweierkomplement:

```
00001010 (10)
11110101 (Einerkomplement)
11110110 (Zweierkomplement)
```

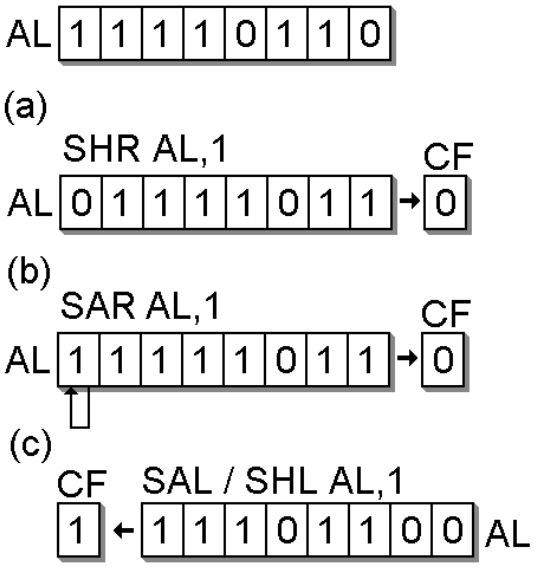

Abb. 6 Abb 1 – Die Schiebebefehle. Der Ausgangszustand ist in der ersten Zeile dargestellt.

Abbildung 1 zeigt nun eine logische und eine arithmetische Rechtsverschiebung. Bei der logischen Rechtsverschiebung geht das Vorzeichen verloren, da eine 0 nachgeschoben wird. Für eine negative Zahl muss deshalb eine arithmetische Verschiebung erfolgen, bei dem immer das *Most Significant Bit* erhalten bleibt (Abbildung 1a). Da in unserem Fall das *Most Significant Bit* eine 1 ist, wird eine 1 nachgeschoben (Abbildung 1b). Ist das *Most Significant Bit* hingegen eine 0, so wird eine 0 nachgeschoben.

Sehen wir uns nun die Linksverschiebung an: Wir gehen wieder von der Zahl –10 aus. Abbildung 1c veranschaulicht arithmetische und logische Linksverschiebung. Wie Sie erkennen können, haben logische und arithmetische Verschiebung keine Auswirkung auf das Vorzeichenbit. Aus diesem Grund gibt es auch zwischen SHL und SAL keinen Unterschied! Beide Befehle sind identisch!

Wie wir mit dem Debugger nachprüfen können, haben beide den selben Opcode:

```
…
sal al,1
shl al,1
…
```

Beim Debugger erhält man:

```
-r
AX=0000  BX=0000  CX=0008  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0D36  ES=0D36  SS=0D36  CS=0D36  IP=0100   NV UP EI PL NZ NA PO NC
0D36:0100 D0E0          SHL    AL,1
-t
```

```
AX=0000  BX=0000  CX=0008  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0D36  ES=0D36  SS=0D36  CS=0D36  IP=0102   NV UP EI PL ZR NA PE NC
0D36:0102 D0E0          SHL    AL,1
```

Seit der 80386-CPU gibt es auch die Möglichkeit den Inhalt von zwei Registern zu verschieben (Abbildung 2). Die Befehle `SHLD` und `SHRD` haben die folgende Syntax:

```
shld / shrd Zieloperand, Quelloperand, Zähleroperand
```

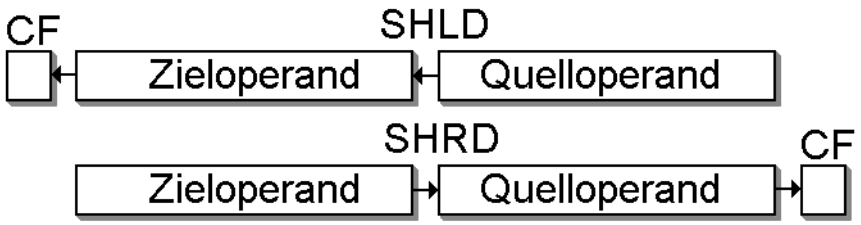

Abb. 7 Abb 2 – Links- und Rechtsshift mit `SHLD` und `SHRD`

Der Zieloperand muss bei beiden Befehlen entweder ein 16- oder 32-Bit-Wert sein, der sich in einer Speicherstelle oder einem Register befindet. Der Quelloperand muss ein 16- oder 32-Bit-Wert sein und darf sich nur in einem Register befinden. Der Zähloperand muss ein Wert oder das CL-Register sein.

5.10 Rotationsbefehle

Bei den Rotationsbefehlen werden die Bits eines Registers ebenfalls verschoben, fallen aber nicht wie bei den Schiebebefehlen an einem Ende heraus, sondern werden am anderen Ende wieder hinein geschoben. Es existieren vier Rotationsbefehle:

- `rol`: Linksrotation
- `ror`: Rechtsrotation
- `rcl`: Linksrotation mit *Carry Flag*
- `rcr`: Rechtsrotation mit *Carry Flag*

Die Rotationsbefehle haben die folgende Syntax:

```
rol / ror /rcl /rcr Zieloperand, Zähleroperand
```

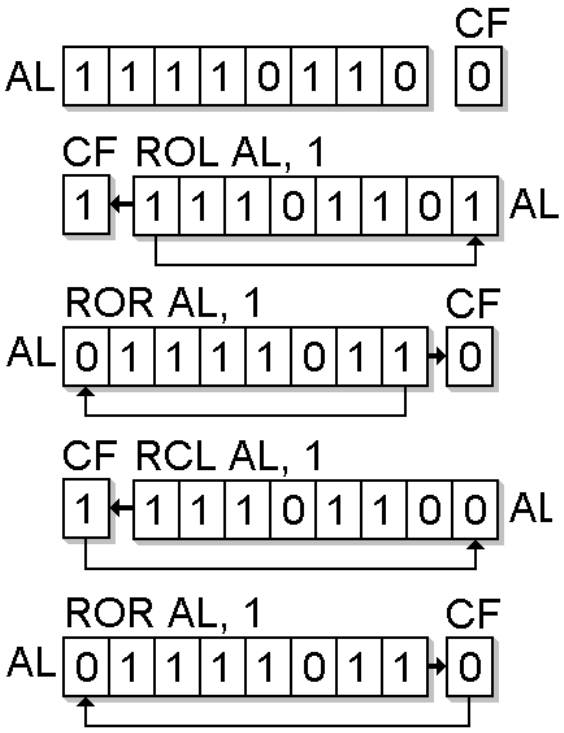

Abb. 8 Abb. 3 – Die Rotationsbefehle. Der Ausgangszustand ist in der ersten Zeile dargestellt.

Der Zähleroperand gibt an, um wie viele Bits der Zieloperand verschoben werden soll. Der Zieloperand kann entweder ein allgemeines Register oder eine Speicherstelle sein. Das Zählerregister kann ein Wert sein oder das CL-Register.

Bei der Rotation mit `rol` und `ror` wird das *Most* bzw. *Least Significant Bit,* das an das andere Ende der Speicherstelle oder des Registers verschoben wird, im *Carry Flag* abgelegt (siehe Abbildung 3).

Die Befehle `rcl` und `rcr` nutzen das *Carry Flag* für die Verschiebung mit: Bei jeder Verschiebung mit dem `rcl`-Befehl wird das *Most Significant Bit* zunächst in das *Carry Bit* verschoben und der Inhalt des *Carry Flags* in das *Least Significant Bit.* Beim `rcr` ist es genau andersherum: Hier wird das *Least Significant Bit* zunächst in das *Carry Flag* geschoben und dessen Inhalt in das *Most Significant Bit* (siehe Abbildung 2).

Wie bei den Schiebebefehlen berücksichtigt der Prozessor (ab 80286) lediglich die oberen 5 Bit des CL-Registers und ermöglicht damit nur eine Verschiebung zwischen 0 und 31.

6 Sprünge und Schleifen

6.1 Unbedingte Sprünge

Unbedingte Sprünge entsprechen im Prinzip einem `GOTO` in höheren Programmiersprachen. Im Assembler heißt der Befehl allerdings `jmp` (Abkürzung für *jump;* dt. „springen“).

Das Prinzip des Sprungbefehls ist sehr einfach: Um die Bearbeitung des Programms an einer anderen Stelle fortzusetzen, wird das IP-Register mit der Adresse des Sprungziels geladen. Da die Adresse des IP-Registers zusammen mit dem CS-Register immer auf den nächsten Befehl zeigt, den der Prozessor bearbeiten wird, setzt er die Bearbeitung des Programms am Sprungziel fort.

Der Befehl hat die folgende Syntax:

```
jmp Zieloperand
```

Je nach Sprungart kann der Zieloperand ein Wert und/oder ein Register bzw. eine Speicherstelle sein. Im Real Mode gibt es drei verschiedene Sprungarten:

- *Short Jump:* Der Sprung erfolgt innerhalb einer Grenze von -128 bis +127 Byte.
- *Near Jump:* Der Sprung erfolgt innerhalb der Grenzen des Code-Segments.
- *Far Jump:* Der Sprung erfolgt innerhalb des gesamten im Real Mode adressierbaren Speichers.

Bevor wir noch genauer auf die Syntax eingehen, sehen wir uns ein Beispiel an, das einen *Short Jump* benutzt. Bei einem *Short Jump* darf der Zieloperand nur als Wert angegeben werden. Ein Register oder eine Speicherstelle ist nicht erlaubt.

Der Wert wird als relative Adresse (bezogen auf die Adresse des nächsten Befehls) interpretiert. Das heißt, dem Inhalt des IP-Registers wird der Wert des Zieloperanden hinzuaddiert. Ist der Wert des Zieloperanden positiv, so liegt das Sprungziel hinter dem `jmp`-Befehl, ist der Wert negativ, so befindet sich das Sprungziel vor dem `jmp`-Befehl.

Das folgende Programm führt einen *Short Jump* aus:

```
org 100h
start:
       jmp short ziel
       ;ein paar unsinnige Befehle:
       mov ax, 05h
       inc ax
       mov bx, 07h
       xchg ax, bx
ziel:  mov cx, 05h
       mov ah,4Ch
       int 21h
```

Das Sprungziel ist durch ein Label gekennzeichnet (hier: `ziel`). Damit kann der Assembler die (absolute oder relative) Adresse für das Sprungziel ermitteln und in den Opcode einsetzen.

Wenn Sie das Programm mit dem Debugger ausführen, bekommen Sie die folgende Ausgabe:

```
-r
AX=0000  BX=0000  CX=0011  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0D3E  ES=0D3E  SS=0D3E  CS=0D3E  IP=0100   NV UP EI PL NZ NA PO NC
0D3E:0100 EB08          JMP     010A
-t
```

Lassen Sie sich nicht von der Ausgabe `JMP 010A` irritieren. Wie Sie am Opcode `EB08` sehen können, besitzt der Zieloperand tatsächlich den Wert `08h`; `EBh` ist der Opcode des (short) `jmp`-Befehls.

Die nächste Adresse nach dem `jmp`-Befehle beginnt mit 0102h. Addiert man hierzu den Wert 08h, so erhält man die Adresse des Sprungziels: 010Ah. Mit dieser Adresse lädt der Prozessor das IP-Register:

```
AX=0000  BX=0000  CX=0011  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=0D3E  ES=0D3E  SS=0D3E  CS=0D3E  IP=010A   NV UP EI PL NZ NA PO NC
0D3E:010A B90500        MOV     CX,0005
-t
```

Wie Sie nun sehen, zeigt nun die Registerkombination CS:IP auf den Befehl `mov cx,0005h`, wo der Prozessor mit der Bearbeitung fortfährt.

Beachten Sie, dass NASM nicht automatisch erkennt, ob es sich um einen *Short* oder *Near Jump* handelt (im Gegensatz zum *Turbo Assembler* und zum *Macro Assembler* ab Version 6.0). Fehlt das Schlüsselwort `short`, so übersetzt dies der Assembler in einen *Near Jump* (was zur Folge hat, dass der Opcode um ein Byte länger ist).

Beim *Short Jump* darf der Zieloperand nur eine relative Adresse besitzen, wohingegen beim Near Jump sowohl eine relative wie auch eine absolute Adresse möglich ist. Dies ist auch leicht einzusehen, wenn Sie sich nochmals den Opcode ansehen: Da nur ein Byte für die Adressinformation zu Verfügung steht, kann der Befehl keine 16bittige absolute Adresse erhalten.

Wenn der *Near Jump*-Befehl aber sowohl ein relative als auch eine absolute Adresse erhalten kann, wie kann der Assembler dann unterscheiden, in welchen Opcode er den Befehl übersetzen muss? Nun, wenn der Zieloperand ein Wert ist, handelt es sich um eine relative Adresse, ist der Zieloperand in einem Register oder einer Speicherstelle abgelegt, so handelt es sich um eine absolute Adresse.

Beim *Far Jump* wird sowohl das CS Register wie auch das IP Register neu geladen. Damit kann sich das Sprungziel auch in einem anderen Segment befinden. Die Adresse kann entweder ein Wert sein oder ein Registerpaar. In beiden Fällen handelt es sich um eine absolute Adresse. Eine relative Adresse existiert beim *Far Jump* nicht.

Das folgende Beispiel zeigt ein Programm mit zwei Codesegmenten, bei dem ein *Far Jump* in das zweite Segment erfolgt:

```
segment code1
  start:
        jmp word far ziel

 segment code2
 ziel:  mov ah, 4Ch
        int 21h
```

Bitte vergessen Sie nicht, dass com-Dateien nur ein Codesegment besitzen dürfen und deshalb das Programm als exe-Datei übersetzt werden muss (siehe Kapitel Das erste Assemblerprogramm / EXE-Dateien[1]).

Das Schlüsselwort word wird benutzt, da es sich um eine 16-Bit-Adresse handelt. Ab der 80386 CPU sind auch 32-Bit-Adressen möglich, da das IP Register 32 Bit breit ist (die Segmentregister sind auch bei der 80386 CPU nur 16 Bit groß).

6.2 Bedingte Sprünge

Mit bedingten Sprüngen lassen sich in Assembler Kontrollstrukturen vergleichbar mit IF ... ELSE Anweisungen in Hochsprachen ausdrücken. Im Gegensatz zum unbedingten Sprung wird dazu nicht in jedem Fall ein Sprung durchgeführt, sondern in Abhängigkeit der Flags.

Natürlich müssen dazu erst die Flags gesetzt werden. Dies erfolgt mit dem cmp- oder test-Befehl.

Der cmp-Befehl hat die folgende Syntax:

```
cmp operand1, operand2
```

Wie Sie sehen, existiert weder ein Quell- noch ein Zieloperand. Dies liegt daran, dass keiner der beiden Operanden verändert wird. Vielmehr entspricht der cmp- einem sub-Befehl, lediglich mit dem Unterschied, dass nur die Flags verändert werden.

Der test-Befehl führt im Gegensatz zum cmp-Befehl keine Subtraktion, sondern eine UND Verknüpfung durch. Der test-Befehl hat die folgende Syntax:

```
test operand1, operand2
```

Mit dem test-Befehl kann der bedingte Sprung davon abhängig gemacht werden, ob ein bestimmtes Bit gesetzt ist oder nicht.

Für Vorzeichenbehaftete Zahlen müssen andere Sprungbefehle benutzt werden, da die Flags entsprechend anders gesetzt sind. Die folgende Tabelle zeigt sämtliche bedingte Sprünge für vorzeichenlose Zahlen:

Sprungbefehl	**Abkürzung für**	**Zustand der Flags**	**Bedeutung**
ja	jump if above	CF = 0 und ZF = 0	Operand1 > Operand2

1 Kapitel 4.4 auf Seite 36

`jae`	jump if above or equal	CF = 0	Operand1 >= Operand2
`jb`	jump if below	CF = 1	Operand1 < Operand2
`jbe`	jump if below or equal	CF = 1 oder ZF = 1	Operand1 <= Operand2
`je`	jump if equal	ZF = 1	Operand1 = Operand2
`jne`	jump if not equal	ZF = 0	Operand1 <> Operand2, also Operand1 != Operand2

Wie Sie an der Tabelle erkennen können, ist jeder Vergleich an eine charakteristische Kombination der *Carry Flags* und des *Zero Flags* geknüpft. Mit dem *Zero Flag* werden die Operanden auf Gleichheit überprüft. Sind beide Operanden gleich, so ergibt eine Subtraktion 0 und das *Zero Flag* ist gesetzt. Sind beide Operanden dagegen ungleich, so ist das Ergebnis einer Subtraktion bei vorzeichenlosen Zahlen einen Wert ungleich 0 und das *Zero Flag* ist nicht gesetzt.

Kleiner und größer werden durch das *Carry Flag* angezeigt: Wenn Zieloperand kleiner als der Quelloperand ist, so wird das *Most Significant Bit* „herausgeschoben“. Dieses Herausschieben wird durch das *Carry Flag* angezeigt. Das folgende Beispiel zeigt eine solche Subtraktion:

```
     0000 1101 (Operand1)
-    0001 1010 (Operand2)
Ü 1 111  1
-------------
   1111 0011
```

Der fett markierte Übertrag stellt den Übertrag dar, der das *Carry Flag* setzt. Wie Sie an diesem Beispiel sehen, lässt sich das Carryflag also auch dazu benutzen um festzustellen, ob der Zieloperand kleiner oder größer als der Quelloperand ist.

Mit diesem Wissen können wir uns nun herleiten, warum die Flags so gesetzt sein müssen und nicht anders:

- `ja`: Hier testen wir mit dem *Carry Flag,* ob die Bedingung Operand1 > Operand2 erfüllt ist. Da die beiden Operanden nicht gleich sein sollen, testen wir außerdem noch das *Zero Flag.*
- `jae`: Der Befehl entspricht dem `ja`-Befehl, mit dem Unterschied, dass wir nicht das *Zero Flag* testen, da beide Operanden gleich sein dürfen.
- `jb`: Hier testen wir mit dem *Carry Flag,* ob die Bedingung Operand1 < Operand2 erfüllt ist. Aber wieso testet `jb` nicht auch das *Zero Flag?* Sehen wir uns dies an einem Beispiel an: Angenommen der erste und zweite Operand haben jeweils den Wert 5. Wenn wir beide voneinander subtrahieren, so erhalten wir als Ergebnis 0. Damit ist das *Zero Flag* gesetzt und das *Carry Flag* ist 0. Folge: Es reicht aus zu prüfen, ob das *Carry Flag* gesetzt ist oder nicht. Die beiden Operanden müssen in diesem Fall ungleich sein.

- `jbe`: Mit dem *Carry Flag* testen wir wieder die Bedingung Operand1 < Operand2 und mit dem *Zero Flag* auf Gleichheit. Beachten Sie, dass entweder das *Carry Flag* **oder** das *Zero Flag* gesetzt werden müssen und nicht beide gleichzeitig gesetzt sein müssen.

Die Befehle `je` und `jne` dürften nun selbsterklärend sein.

Wie bereits erwähnt, unterscheiden sich die bedingten Sprünge mit vorzeichenbehafteten Zahlen von Sprüngen mit vorzeichenlosen Zahlen, da hier die Flags anders gesetzt werden:

Sprungbefehl	**Abkürzung für**	**Zustand der Flags**	**Bedeutung**
`jg`	jump if greater	ZF = 0 und SF = OF	Operand1 > Operand2
`jge`	jump if greater or equal	SF = OF	Operand1 >= Operand2
`jl`	jump if less	ZF = 0 und SF <> OF	Operand1 < Operand2
`jle`	jump if less or equal	SF = 1 oder SF <> OF	Operand1 <= Operand2

Die Befehle `je` und `jne` sind bei vorzeichenlosen und vorzeichenbehafteten Operatoren identisch und tauchen deshalb in der Tabelle nicht noch einmal auf.

Darüber hinaus existieren noch Sprungbefehle, mit denen sich die einzelnen Flags abprüfen lassen:

Sprungbefehl	**Abkürzung für**	**Zustand der Flags**
`jc`	jump if carry	CF = 1
`jnc`	jump if not carry	CF = 0
`jo`	jump if overflow	OF = 1
`jno`	jump if not overflow	OF = 0
`jp`	jump if parity	PF = 1
`jpe`	jump if parity even	PF = 1
`jpo`	jump if parity odd	PF = 0

Es fällt auf, dass sowohl der `jb` und `jae` wie auch `jc` und `jnc` testen, ob das *Carry Flag* gesetzt bzw. gelöscht ist. Wie kann der Prozessor also beide Befehle unterscheiden? Die Antwort lautet: Beide Befehle sind Synonyme. Die Befehle `jb` und `jc` besitzen beide den Opcode 72h, `jae` und `jnc` den Opcode 73h. Das gleiche gilt auch für `jp` und `jpe`, die ebenfalls Synonyme sind.

Das folgende Beispiel zeigt eine einfache `IF-THEN`-Anweisung:

```
org 100h
start:
  mov dx,meldung1
  mov ah,9h
  int 021h
  mov ah, 01h    ;Wert über die Tastatur einlesen
  int 021h
  cmp al, '5'
  jne ende
```

```
  mov dx,meldung2
  mov ah,9h
  int 021h
ende:
  mov ah,4Ch
  int 21h
section .data
  meldung1: db 'Bitte geben Sie eine Zahl ein:', 13, 10, '$'
  meldung2: db 13, 10, 'Sie haben die Zahl 5 eingegeben.', 13, 10, '$'
```

Nachdem Sie das Programm gestartet haben, werden Sie aufgefordert, eine Zahl einzugeben. Wenn Sie die Zahl 5 eingeben, erscheint die Meldung `Sie haben die Zahl 5 eingeben.`

Wie Sie sehen, prüfen wir nicht Gleichheit, sondern auf Ungleichheit. Der Grund hierfür ist, dass wir die nachfolgende Codesequenz ausführen wollen, wenn der Inhalt des AL Registers 5 ist. Da wir aber nur die Möglichkeit eines bedingten Sprungs haben, müssen wir überprüfen ob AL ungleich 5 ist.

Das folgende Programm zeigt die Implementierung einer `IF-THEN-ELSE`-Anweisung im Assembler:

```
org 100h
start:
    mov dx,meldung1
    mov ah,9h
    int 021h
    mov ah, 01h    ;Wert über die Tastatur einlesen
    int 021h
    cmp al, '5'

    ja l1
    mov dx,meldung2
    mov ah,9h
    int 021h
    jmp ende

l1: mov dx,meldung3
    mov ah,9h
    int 021h
ende:
    mov ah,4Ch
    int 21h
section .data
meldung1: db 'Bitte geben Sie eine Zahl ein:', 13, 10, '$'
meldung2: db 13, 10, 'Sie haben eine Zahl kleiner oder gleich 5 eingegeben',
13, 10, '$'
meldung3: db 13, 10, 'Sie haben eine Zahl groesser 5 eingegeben' , 13, 10, '$'
```

Auch hier besitzt der bedingte Sprung eine „umgekehrte Logik“. Beachten Sie den unbedingten Sprung. Er verhindert, dass auch der `ELSE`-Zweig des Programms ausgeführt wird und darf deshalb nicht weggelassen werden.

6.3 Schleifen

Wie in jeder anderen Sprache auch hat der Assembler Schleifenbefehle. Eine mit FOR vergleichbare Schleife lässt sich mit `loop` realisieren. Der `loop`-Befehl hat die folgende Syntax:

```
loop adresse
```

Bei der Adresse handelt es sich um eine relative Adresse, die 8 Bit groß ist. Da das Offset deshalb nur einen Wert von −128 bis +127 einnehmen darf, kann eine Schleife maximal 128 Byte Code erhalten – dies gilt auch für eine 32-Bit-CPU wie beispielsweise den Pentium IV. Umgehen lässt sich diese Beschränkung, wenn man die Schleife mit den Befehlen der bedingten Sprünge konstruiert, beispielsweise mit einem *Near Jump*.

Die Anzahl der Schleifendurchgänge befindet sich immer im CX-Register (daher auch der Name *„Counter Register“*). Bei jedem Durchgang wird der Inhalt um 1 verringert. Ist der Inhalt des CX-Registers 0 wird die Schleife verlassen und der nächste Befehl bearbeitet. Der Befehl ließe sich also auch mit der folgenden Befehlsfolge nachbilden:

```
dec cx
jne adresse
```

Wenn der Schleifenzähler um mehr als 1 verringert werden soll, kann der `sub`-Befehl verwendet werden. Wenn der Schleifenzähler um 2 oder 3 verringert wird, werden die meisten Assemblerprogrammierer stattdessen auf den `dec`-Befehl zurückgreifen, da dieser schneller als der `sub`-Befehl ist.

Das nachfolgende Programm berechnet 7!, die Fakultät von 7:

```
org 100h
start:
  mov cx, 7
  mov ax, 1
  mov bx, 1
schleife:
  mul bx
  inc bx
  loop schleife
  mov ah, 4Ch
  int 21h
```

Mit dem Befehl `mov cx, 7` wird der Schleifenzähler initialisiert. Insgesammt wird die Schleife 7-mal durchlaufen, im ersten Durchlauf mit CX=7 und BX=1, im letzten mit CX=1 und BX=7.

Das Ergebnis kann mit jedem Debugger überprüft werden (AX=13B0).

Neben den Normalen `FOR`-Schleifen existieren in den meisten Hochsprachen auch noch `WHILE`- oder „`REPEAT ... UNTIL`“-Schleifen. Im Assembler kann dazu die `loope` (loop if equal) und `loopne` (loop if not equal) als Befehl benutzt werden. Beide Befehle prüfen nicht nur, ob das CX-Register den Wert 0 erreicht hat, sondern zusätzlich noch das Zeroflag: Der `loope` durchläuft die Schleife solange, wie der Zeroflag gesetzt ist, der `loopne`-Befehl dagegen, solange das Zeroflag ungleich 0 ist.

7 Unterprogramme und Interrupts

7.1 Der Stack

Bevor wir uns mit Unterprogrammen und *Interrupts* beschäftigen, müssen wir uns noch mit dem Stack (dt. „Stapel“) vertraut machen. Der Stack ist ein wichtiges Konzept bei der Assemblerprogrammierung.

Dazu stellen wir uns vor, wir wären Tellerwäscher in einem großen Hotel. Unserer Aufgabe besteht neben dem Waschen der Teller auch im Einräumen derselbigen ins Regal. Den erste Teller, den wir dort stapeln, bezeichnen wir als Teller 1, den nächsten als 2 usw. Wir nehmen an, dass wir 5 Teller besitzen.

Wenn wir nun wieder einen Teller benötigen, nehmen wir den obersten Teller vom Stapel. Das ist bei uns der Teller mit der Tellernummer 5. Benötigen wir erneut einen, dann nehmen wir Teller 4 vom Stapel – und nicht etwa Teller 1, 2 oder 3. Beim nächsten Waschen stellen wir zunächst Teller 4 und anschließend 5 (oder umgekehrt) auf den Stapel. Wir nehmen dabei immer den obersten Teller und nie einfach einen Teller aus der Mitte.

Das Konzept bei einem Stapel des Prozessors ist ganz ähnlich: Wir können immer nur auf das oberste Element des Stapels zugreifen. Ein Element können wir mit dem `push`-Befehl auf den Stapel legen. Mit dem `pop`-Befehl holen wir uns das oberste Element wieder vom Stapel.

Das Prinzip, dass wir das Element vom Stack holen, das wir zuletzt auf den Stapel gelegt haben, bezeichnet man als LIFO–Prinzip. Dies ist die englische Abkürzung für *Last In First Out* (dt. „zuletzt herein – zuerst heraus“).

Das folgende Programm legt 3 Werte auf den Stack und holt diese anschließend wieder vom Stapel:

```
segment stack stack
       resb 64h

segment code

  ..start:
  mov ax, 0111h
  mov bx, 0222h
  mov cx, 0333h
  push ax
  push bx
  push cx
  pop bx
  pop ax
  pop cx
  mov ah, 4Ch
  int 21h
```

Neben den Befehlen `pop` und `push` sind auch die ersten beiden Assembleranweisungen neu hinzugekommen. Mit diesen beiden Anweisungen wird ein Stack von 64 Byte (Hexadezimal) angelegt.

Diese Information erhält auch der Kopf der EXE-Datei. Deshalb kann das Betriebssystem den Register SP (Stack Pointer) mit der Größe des Stacks initialisieren:

```
AX=0111  BX=0222  CX=0333  DX=0000  SP=0064  BP=0000  SI=0000  DI=0000
DS=0D3A  ES=0D3A  SS=0D4A  CS=0D51  IP=0009   NV UP EI PL NZ NA PO NC
0D51:0009 50          PUSH    AX
```

Die Register SS *(Stack Segment)* und SP erhalten zusammen einen Zeiger, der auf das oberste Element des Stacks zeigt.

Der Push Befehl legt nun den Inhalt des Registers auf den Stack. Wir wollen uns nun ansehen, wie sich der Inhalt der Register verändert.

```
AX=0111  BX=0222  CX=0333  DX=0000  SP=0062  BP=0000  SI=0000  DI=0000
DS=0D3A  ES=0D3A  SS=0D4A  CS=0D51  IP=000A   NV UP EI PL NZ NA PO NC
0D51:000A 53          PUSH    BX
```

Beachten Sie bitte, dass sich der Stack in Richtung kleinere Adressen wächst. Daher erhalten wir als neue Zeiger für den SP-Register 0062 und beim nächsten `push` den Wert 0060 im SP-Register:

```
-t

AX=0111  BX=0222  CX=0333  DX=0000  SP=0060  BP=0000  SI=0000  DI=0000
DS=0D3A  ES=0D3A  SS=0D4A  CS=0D51  IP=000B   NV UP EI PL NZ NA PO NC
0D51:000B 51            PUSH    CX
-t

AX=0111  BX=0222  CX=0333  DX=0000  SP=005E  BP=0000  SI=0000  DI=0000
DS=0D3A  ES=0D3A  SS=0D4A  CS=0D51  IP=000C   NV UP EI PL NZ NA PO NC
0D51:000C 59            POP     BX
-t
```

Anschließend holen wir die Werte wieder vom Stack. Der `pop`-Befehl holt den obersten Wert vom Stack und speichert dessen Inhalt im Register BX. Außerdem wird der *Stack Pointer* (SP-Register) um zwei erhöht. Damit zeigt er nun auf das nächste Element auf unserem Stack:

```
AX=0111  BX=0333  CX=0333  DX=0000  SP=0060  BP=0000  SI=0000  DI=0000
DS=0D3A  ES=0D3A  SS=0D4A  CS=0D51  IP=000D   NV UP EI PL NZ NA PO NC
0D51:000D 58          POP     AX
-t
```

In unserem Tellerbeispiel entspricht dies dem obersten Teller, den wir vom Stapel genommen haben. Da wir zuletzt den Wert 0333 auf den Stack gelegt haben, wird dieser Wert nun vom `pop`-Befehl vom Stack geholt.

Anschließend holen wir auch die anderen beiden Werte vom Stack. Der Vorgang ist analog, und sie können dies daher selbst mit Hilfe des Debuggers wenn nötig nachvollziehen.

Auch com-Programme können übrigens einen Stack besitzen. Wie sie ja bereits wissen, können diese maximal ein Segment groß werden. Daher befinden sich Code, Daten und Stack in einem einzigen Segment. Damit diese nicht in Konflikt miteinander geraten, wird der SP Register vom Betriebssystem mit dem Wert FFFE geladen. Da der Stack in Richtung negativer Adressen wächst, kommt es nur dann zu einem Konflikt, wenn Code, Daten und Stack zusammen eine Größe von mehr als 64 kB haben.

7.2 Unterprogramme

Unterprogramme dienen zur Gliederung des Codes und damit zur besseren Lesbarkeit und Wartbarkeit. Ein Unterprogramm ist das Äquivalent zu Funktionen und / oder Prozeduren in prozeduralen Sprachen wie C, Pascal oder Ada. Die Compiler prozeduraler Hochsprachen dürften fast ausnahmslos Funktionen bzw. Prozeduren in Assembler-Unterprogramme übersetzen (abgesehen von einigen Ausnahmen wie beispielsweise *inline*-Funktionen in C, mit dem man sich einen Funktionsaufruf ersparen kann – wenn es der Compiler denn für sinnvoll hält).

Bei (rein) objektorientierten Sprachen hingegen existiert kein Äquivalent zu Unterprogrammen. Eine gewisse Ähnlichkeit haben jedoch Operationen bzw. Methoden, die allerdings immer Bestandteil einer Klasse sein müssen.

Ein Unterprogramm wird wie normaler Code ausgeführt; der Code wird aber nach Beenden des Unterprogramms an der Stelle fortgesetzt, von wo das Unterprogramm aufgerufen wurde. Zum Aufrufen eines Unterprogrammes dient der Befehl `call`. Dieser schiebt seine Aufrufadresse auf den Stack und ruft das Unterprogramm anhand des Labels auf. Das Unterprogramm benutzt die Aufrufadresse auf dem Stack, um wieder zu der aufrufenden Stelle zurück zu springen.

Der `call`-Befehl hat die folgende Syntax:

```
call Sprungmarke
```

Wie beim `jmp`-Befehl existieren mehrere Arten des `call`-Befehls: Beim `near-call` liegt das Sprungziel innerhalb eines Segments; beim `far-call` wird zusätzlich der Segmentregister herangezogen, so dass das Sprungziel auch in einem anderen Segment liegen kann. Außerdem kann der `near-call`-Befehl sowohl eine relative Adresse wie auch eine absolute Adresse besitzen. Die relative Adresse muss als Wert angegeben werden, wohingegen sich die absolute Adresse in einem Register oder in einer Speicherstelle befinden muss. `Der far-call`-Befehl hingegen kann nur eine absolute Adresse besitzen.

Dies ist alles zugegebenermaßen sehr unübersichtlich. Daher soll die folgende Tabelle nochmals eine Übersicht über die verschiedenen Arten des `call`-Befehls geben:

Befehl (Intel Syntax)	**NASM Syntax**	**Beschreibung**
`call rel16`	`call imm`	Near-Call, relative Adresse
`call r/m16`	`call r/m16`	Near-Call, absolute Adresse
`call ptr16:16`	`call imm:imm16`	Far-Call, absolute Adresse

`call m16:16`	`call far mem16`	Far-Call, absolute Adresse

Die hier verwendete Schreibweise für die linke Spalte ist aus dem *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference* übernommen, die Schreibweise für die mittlere Spalte dem NASM Handbuch. Die Abkürzung `rel16` bedeutet, dass der Operand eine relative 16 Bit Adresse ist, die als Zahlenwert direkt hinter dem Befehl folgt, `r/m16` bedeutet, dass sich der Operand in einem der allgemeinen 16-Bit Register (AX, BX, CX, DX, SP, BP, SI, DI) oder einer 16-Bit breiten Speicherstelle befinden muss. Wie dann leicht zu erraten, bedeutet `m16`, dass sich der Operand nur in einer 16-Bit Speicherstelle befinden darf. Beim `call-ptr16:16`-Befehl besteht der Zeiger auf die Zieladresse aus einem Segmentanteil, der sich in einem der Segmentregister befindet und einem Offsetteil, der sich in einem der allgemeinen Register befindet.

Die NASM Schreibweise ist ganz ähnlich. `Imm16` ist steht für *immediate value* und ist ein Operand, der direkt als Wert übergeben wird. Bei NASM müssen Sie lediglich angeben, ob es sich um einen `far-call` handelt oder nicht.

Das ganze sieht komplizierter aus als es in der Praxis ist: Die Hauptaufgabe übernimmt der Assembler, der das Label entsprechend übersetzt.

Zum Zurückkehren aus dem Unterprogramm in das Hauptprogramm wird der `ret`-Befehl benutzt. Er hat die folgende Syntax:

```
ret [Stackanzahl]
```

Optional kann man den Parameter `Stackanzahl` angeben, der bestimmt, wie viele Bytes beim Zurückspringen von dem Stack abgezogen werden sollen. Diese Möglichkeit wird verwendet, wenn man Parameter zur Übergabe an das Unterprogramm gepusht hat und sich nach dem Rücksprung das poppen ersparen möchte (siehe weiter unten[1]).

Auch für den `ret`-Befehl gibt es eine `far-` und eine `near`-Variante. Beide Varianten unterscheiden sich allerdings nur im Opcode. Der Assembler übersetzt den Befehl automatisch in die richtige Variante. Für den Programmierer ist die Unterscheidung deshalb nicht so sehr von Bedeutung. Für den Prozessor aber sehr wohl, da er wissen muss, ob er im Fall eines `near-calls` nur das oberste Element des Stacks in den IP Register zurückschreiben muss, oder im Fall eines `far-calls` die zwei obersten Elemente in die Register CS und IP. Auf jeden `far-call` muss deshalb ein entsprechender `far-ret` kommen und auf einen `near-call` ein entsprechender `near-ret`.

7.3 Beispiele für Unterprogramm-Aufrufe

Zwei Beispiele, aufbauend auf dem ersten Programm „Hello World!“ vom Beginn dieses Buchs, sollen das Prinzip illustrieren. Im ersten Beispiel steht das Unterprogramm in derselben Quelldatei wie das Hauptprogramm (ähnlich dem Beispiel oben in diesem Abschnitt). Im zweiten Beispiel steht das Unterprogramm in einer separaten Datei, die dem Unterprogramm vorbehalten ist.

1 `http://de.wikibooks.org/wiki/unten`

Wenn das Unterprogramms nicht von mehreren, sondern nur von einem einzigen Programm verwendet wird, stehen beide Codeteile, Haupt- und Unterprogramm, besser in der selben Quelldatei. Das zeigt das erste der folgenden zwei Beispiele. Falls das Unterprogramm jedoch für mehrere Programme nützlich ist, ist es ökonomisch, es in eine eigene Quelldatei zu schreiben und diese bei Bedarf immer wieder mit einem anderen Programm zu verwenden. Das Linker-Programm verbindet die beiden Dateien zu einem gemeinsamen Programm.

1. Beispiel: Haupt- und Unterprogramm stehen in der selben Quelldatei

Aufbauend auf dem Beispiel der Exe-Datei des *Hello-World*-Programms ändern wir dessen Quelltext in:

```
segment code

..start:
        mov ax, data
        mov ds, ax
        call print
        mov ah, 4Ch
        int 21h

 print: mov dx, hello
        mov ah, 09h
        int 21h
        ret

segment data
 hello: db 'Hello World!', 13, 10, '$'
```

Assemblieren und linken Sie diesen Quellcode wie gewohnt mit Hilfe der folgenden zwei Befehle. Nennen Sie das Programm z. B. „hwrldsub“:

```
nasm hwrldsub.asm  -fobj -o hwrldsub.obj
alink hwrldsub.obj
```

(Die Warnung des Linkers alink „no stack“ können Sie in diesen Beispielen ignorieren.) Starten Sie das daraus gewonnene Programm hwrldsub.exe. Sie sehen, dass das Programm erwartungsgemäß nichts weiter macht, als „Hello World!“ auf dem Bildschirm auszugeben.

Im Quelltext haben wir den Codeteil, der den Text ausgibt, einfach an den Fuß des Codes verlagert, und diesen Block zwischen dem Unterprogramm-Namen `print` und den Rückkehrbefehl `ret` eingeschlossen. An der Stelle, wo dieser Code-Teil zuvor stand, haben wir den Befehl zum Aufruf des Unterprogramms `call` und den Namen des Unterprogramms `print` eingefügt. Der Name des Unterprogramms ist mit wenigen Einschränkungen frei wählbar.

Gelangt das Programm an die Zeile mit dem Unterprogramm-Aufruf `call`, wird das IP-Register zunächst erhöht, anschließend auf dem Stack gespeichert und schließlich das Programm am Label `print` fortgesetzt. Der Code wird weiter bis zum `ret`-Befehl ausgeführt. Der `ret`-Befehl holt das oberste Element vom Stack und lädt den Wert in den IP-Register, so dass das Programm am Befehl `mov ah, 4Ch` fortgesetzt wird, das heißt er ist zurück ins Hauptprogramm gesprungen.

Wir empfehlen zum besseren Verständnis eine Untersuchung des Programmablaufs mit dem Debugger. Achten Sie hierbei insbesondere auf die Werte im IP- und SP-Register.

Lassen Sie sich zur Untersuchung der Abläufe neben dem Fenster, in dem Sie mit `debug` das Programm Schritt für Schritt ausführen, in einem zweiten Fenster anzeigen, wie der Assembler den Code des Programms in den Speicher geschrieben hat. Dazu deassemblieren Sie das Programm `hwrldsub.exe` mit dem `debug`-Programm. Geben Sie dazu ein:

```
debug hwrldsub.exe
-u
```

Das Ergebnis ist ein Programm-Listing. Vergleichen Sie zum Beispiel den Wert des Befehlszeigers bei jedem Einzelschritt mit den Programmzeilen des Programm-Listings.

In der Praxis wird wird man einen Codeteil, der wie in diesem Beispiel nur ein einziges Mal im Programm läuft, allerdings nicht in ein Unterprogramm schreiben; das lohnt nicht den Aufwand. In diesem Punkt ist unser Beispiel theoretisch und stark vereinfacht; es soll nur das Prinzip des Aufrufs zeigen. Lohnend, das heißt Tipparbeit sparend und Übersicht schaffend, ist ein Unterprogramm, wenn der Code darin voraussichtlich mehrmals während des Programmlaufs nötig ist.

2. Beispiel: Haupt- und Unterprogramm stehen in getrennten Quelldateien

Speichern und assemblieren Sie die Quelldatei

```
extern print

segment code
..start:
  mov ax, daten
  mov ds, ax
  mov dx, hello

  call far print

  mov ah, 4ch
  int 21h

segment daten
  hello: db 'Hello World!', 13, 10, '$'
```

zum Beispiel mit dem Namen `mainprog`, und die Quelldatei

```
global print

segment code

print:  mov ah, 09h
        int 21h
        retf
```

zum Beispiel mit dem Namen `sub_prog.asm`

Kompilieren sie anschließend beide Programme, so dass Sie die Dateien `mainprog.obj` und `sub_prog.obj` erhalten. Anschließend verbinden (linken) Sie beide mit dem Befehl

```
alink mainprog.obj sub_prog.obj
```

Das Ergebnis, die ausführbare Exe-Datei, trägt den Namen mainprog.exe und produziert erwartungsgemäß den Text „Hello World!“ auf dem Bildschirm. (An diesem Beispiel können Sie auch sehr gut erkennen, warum der Assembler bzw. der Linker bei Exe-Dateien die Anweisung `..start` erwartet: So weiß der Linker, wie er die Dateien richtig zusammenlinken muss.)

Die Besonderheiten dieses Unterprogrammaufrufs im Unterschied zum ersten Beispiel:

- `extern print` erklärt dem Assembler, dass er die Programm-Marke `print` in einer anderen Quelldatei findet, obwohl sie in dieser Quelldatei eingeführt (definiert) wird.
- `far` im Aufruf `call` des Unterprogramms macht dem Assembler klar, dass sich der Code für das Unterprogramm in einer anderen Quelldatei befindet.
- `global print` ist das Pendant zu `extern print`: Es zeigt dem Assembler, dass der Aufruf des Codeteils `print` in einer anderen Quelldatei steht.
- `retf`, das „f“ steht für „far“, erklärt dem Assembler, dass das Ziel des Rücksprungs in einer anderen Quelldatei zu finden ist.

Diese Art des Unterprogramms ist nützlich, wenn man die Funktion des Unterprogramms in mehreren Hauptprogrammen benötigt. Dann schreibt und assembliert man die Quelldatei mit dem Unterprogramm nur einmal, bewahrt Quell- und Objektdatei gut auf und verlinkt die Objektdatei jedes gewünschten Hauptprogramms mit der des Unterprogramms.

8 Stringbefehle

8.1 Maschinensprache-Befehle zur Stringverarbeitung

Die CPU der 80x86-Familie besitzen spezielle Befehle, mit denen die Verarbeitung von Massendaten, zum Beispiel von Zeichenketten, optimiert werden kann.

8.1.1 Beispiel: Kopieren Byteweise ab Adresse SI nach DI

```
        ;
        MOV         CX,40    ; Zeichenanzahl
SCHL:   MOV         AL,[SI]
        MOV         [DI],AL
        INC         SI
        INC         DI
        LOOP        SCHL
```

In jedem Schleifendurchlauf muss der Prozessor die gleichen fünf Befehle immer neu lesen und decodieren. Ersetzt man obige Befehle durch

```
        ;
        MOV         CX,40    ; Zeichenanzahl
        REP MOVSB
```

wird das Programm kürzer und läuft auch viel schneller.

Hierbei gibt es folgende Befehle:

Übertragungsbefehle:

`MOVSx*`	Kopiert ein Zeichen von der Adresse ds:si/esi an die Adresse es:di/edi und inkrementiert/dekrementiert** si/esi und di/edi um die Zeichengröße.
`LODSx*`	Kopiert ein Zeichen von der Adresse ds:si/esi in den Akkumulator (al/ax/eax) und inkrementiert/dekrementiert** si/edi um die Zeichengröße.
`STOSx*`	Kopiert ein Zeichen aus dem Akkumulator an die Adresse es:di/edi und inkrementiert/dekrementiert** di/edi um die Zeichengröße.

Vergleichsbefehle:

`SCASx*`	Vergleicht ein Zeichen von der Adresse ds:si/esi mit den Akkumulator (al/ax/eax) und setzt die Flags wie cmp. Inkrementiert/dekrementiert** di/edi um die Zeichengröße.

Übertragungsbefehle:	
`CMPSx`*	Vergleicht ein Zeichen von der Adresse ds:si/esi mit einem von der Adresse es:di/edi und setzt die Flags wie cmp. Inkrementiert/dekrementiert** si/esi und di/edi um die Zeichengröße.
Wiederholungspräfixbefehle:	
`REP`, `REPE` oder `REPZ`	Führt den folgenden Stringbefehl solange durch und dekrementiert jedes mal cx/ecx um 1, bis cx/ecx null erreicht oder die Zero-Flag nicht gesetzt ist.
`REPNE` oder `REPNZ`	Führt den folgenden Stringbefehl solange durch und dekrementiert jedes mal cx/ecx um 1, bis cx/ecx null erreicht oder die Zero-Flag gesetzt ist.

* x symbolisiert hierbei die Zeichengröße. `B` steht für einen Byte, `W` für ein Wort und `D` (im 32 bit Modus) für ein Doppelwort.** Wenn die Direktion Flag gelöscht gelöscht ist (erreichbar mit CLD), wird inkrementiert, ist sie gesetzt (erreichbar mit STD) wird dekrementiert.Ob

8.1.2 Beispiel: Das erste Leerzeichen in einem String finden

```
      ;
      MOV    CX,SIZE STRING   ; Stringlänge
      CLD                     ; Richtung
      MOV    AL,20H           ; Quelloperand: Leerzeichen

      MOV    DI,SEG STRING    ; Segmentadresse
      MOV    ES,DI
      MOV    DI,OFFSET STRING ; Offsetadresse

      REPNE SCASB             ; Wiederholen solange ungleich AL

      JNZ    xxx              ; bis Ende durchsucht und kein Leerzeichen
gefunden
      JZ     yyy              ; bis Ende durchsucht und Leerzeichen gefunden
```

8.1.3 Beispiel: Zwei Strings auf Gleichheit überprüfen

```
      ;
      CLD                      ; In Richtung steigender Adressen vergleichen
      MOV    CX,SIZE STRING1   ; Stringlänge

      MOV    SI,SEG STRING1    ; Segmentadresse
      MOV    DS,SI
      MOV    SI,OFFSET STRING1 ; Offsetadresse

      MOV    DI,SEG STRING2    ; Segmentadresse
      MOV    ES,DI
      MOV    DI,OFFSET STRING2 ; Offsetadresse

      REPE CMPSB               ; Wiederholen solange ungleich AL

      JNZ   xxx                ; Strings sind ungleich
      JZ    yyy                ; Strings sind gleich
```

9 Befehlsreferenz

Dieses Kapitel beschreibt die Assembler-Syntax in der Intelschreibweise.

Die Syntaxbeschreibungen bestehen jeweils aus drei Teilen: Dem Opcode in hexadezimal, der eigentlichen Syntax und dem Prozessor, seit dem der Befehl existiert. Der Opcode-Aufbau soll in Zukunft in einem eigenen Kapitel beschrieben werden. Die Syntax hat die folgenden Bedeutungen:

- *r8:* Eines der folgenden 8-Bit-Register kann verwendet werden: AH, AL, BH, BL, CH, CL, DH oder DL
- *r16:* Eines der folgenden 16-Bit-Register kann verwendet werden: AX, BX, CX oder DX; BP, SP, DI oder SI
- *r32:* Eines der folgenden 32-Bit-Register kann verwendet werden: EAX, EBX, ECX oder EDX; EBP, ESP, EDI oder ESI
- *imm8:* Ein Bytewert zwischen –128 bis +127
- *imm16:* Ein Wortwert zwischen –32.768 bis +32.767
- *r/m8:* Der Wert kann entweder ein allgemeines 8-Bit-Register (AH, AL, BH, BL, CH, CL, DH oder DL) oder ein Bytewert aus dem Arbeitsspeicher sein
- *r/m16:* Der Wert kann entweder ein allgemeines 16-Bit-Register (AX, BX, CX oder DX; BP, SP, DI oder SI) oder ein Bytewert aus dem Arbeitsspeicher sein
- *SReg:* Segmentregister

Der Aufbau des Opcodes hat die folgende Bedeutung:

- *ib:* Operation bezieht sich auf 8 Bit Daten. (**b**yte)
- *iw:* Operation bezieht sich auf 16 Bit Daten. (**w**ord)
- *id:* Operation bezieht sich auf 32 Bit Daten. (**d**ouble word)
- */0* bis */7, /r, r+b, r+w, r+d:* Interne Information zum Aufbau des Maschinenbefehls

Wenn nicht explizit etwas anderes angegeben ist, haben die Flags die folgende Bedeutung:

- OF – *Overflow Flag* / Überlauf
- SF – *Sign Flag* / Vorzeichen
- ZF – *Zero Flag* / Ist 1 falls das Resultat 0 ist, sonst 0
- AF – *Auxiliary Flag* / Hilfs-Übertragsflag bei Übertrag von Bit 3 auf 4. Dies macht nur Sinn bei BCD-Zahlen
- PF – *Parity Flag* / Paritätsflag
- CF – *Carry Flag* / Übertrag

9.1 ADD (Add)

ADD addiert zu einem Speicherbereich oder einem Register einen festen Wert oder den Wert eines Registers.

```
04 ib       add AL, imm8       8086+
05 iw       add AX, imm16      8086+
80 /0 ib    add r/m8, imm8     8086+
81 /0 iw    add r/m16, imm16   8086+
83 /0 ib    add r/m16, imm8    8086+
00 /r       add r/m8, r8       8086+
01 /r       add r/m16, r16     8086+
02 /r       add r8, r/m8       8086+
03 /r       add r16, r/m16     8086+
05 id       add eax,imm32      80386+
81 /0 id    add r/m32, imm32   80386+
83 /0 ib    add r/m32, imm8    80386+
01 /r       add r/m32, r32     80386+
03 /r       add r32, r/m32     80386+
```

Flags:

- OF, SF, ZF, AF, CF, PF

Beispiel:

```
ADD eax,10
ADD eax,ebx
```

9.2 ADC (Add with carry)

ADC addiert zu einem Speicherbereich oder einem Register einen festen Wert sowie das Carry-Flag oder den Wert eines Registers.

```
14 ib       adc AL, imm8       8086+
15 iw       adc AX, imm16      8086+
80 /2 ib    adc r/m8, imm8     8086+
81 /2 iw    adc r/m16, imm16   8086+
83 /2 ib    adc r/m16, imm8    8086+
10 /r       adc r/m8, r8       8086+
11 /r       adc r/m16, r16     8086+
12 /r       adc r8, r/m8       8086+
13 /r       adc r16, r/m16     8086+
15 id       adc EAX,imm32      80386+
83 /2 ib    adc r/m32, imm8    80386+
11 /r       add r/m32, r32     80386+
13 /r       add r32, r/m32     80386+
```

Flags:

- OF, SF, ZF, AF, CF, PF

9.3 AND

Verknüpft die Operanden durch ein logisches AND.

```
24 ib       and AL, imm8       8086+
25 iw       and AX, imm16      8086+
80 /4 ib    and r/m8, imm8     8086+
81 /4 iw    and r/m16, imm16   8086+
83 /4 ib    and r/m16, imm8    8086+
20 /r       and r/m8, r8       8086+
21 /r       and r/m16, r16     8086+
22 /r       and r8, r/m8       8086+
23 /r       and r16, r/m16     8086+
25 id       and EAX, imm32     80386+
81 /4 id    and r/m32, imm32   80386+
21 /r       and r/m32, r32     80386+
83 /4 ib    and r/m32, imm8    80386+
23 /r       add r32, r/m32     80386+
```

Flags:

- OF und CF werden gelöscht
- SF, ZF und PF
- AF ist undefiniert

9.4 CLC (Clear Carry Flag)

Löscht das Carry-Flag.

```
F8          clc                8086+
```

Flags:

- CF wird gelöscht

9.5 CLI (Clear Interrupt Flag)

Löscht das Interrupt-Flag. Die CPU bearbeitet keine Interrupt-Requests (Hardware-Interrupts) mehr.

```
FA          cli                8086+
```

Flags:

- IF wird gelöscht

9.6 CMC (Complement Carry Flag)

Dreht den Wahrheitswert des Carry-Flags um.

```
F4          cmc             8086+
```

Flags:

- CF -Wert wird invertiert; [0==>1 | 1==>0]

9.7 DEC (Decrement)

Subtrahiert 1 vom Zieloperanden.

```
FE /1       dec r/m8        8086+
FF /1       dec r/m16       8086+
48 r+w      dec r16         8086+
FF /1       dec r/m32       80386+
48 r+w      dec r32         80386+
```

Flags:

- OF, SF, ZF, AF, PF
- Das Carry-Flag wird nicht beeinflust

Beispiel:

```
DEC eax
DEC [eax]
```

9.8 DIV (Unsigned Divide)

Dividiert das AX-, DX:AX- oder EDX:EAX-Register durch den Quelloperanden und speichert das Ergebnis im AX-, DX:AX- oder EDX:EAX-Register.

```
F6 /6       div r/m8        8086+
F7 /6       div r/m16       8086+
F7 /6       div r/m32       80386+
```

Flags:

- CF, OF, SF, ZF, AF und PF sind undefiniert

9.9 IMUL (Signed Multiply)

Multipliziert eine vorzeichenbehaftete Zahl.

```
F6 /5      imul r/m8              8086+
F7 /5      imul r/m16             8086+
F7 /5      imul r/m32             80386+
0F AF /r   imul r16, r/m16        80386+
0F AF /r   imul r32, r/m32        80386+
6B /r ib   imul r16, r/m16, imm8  80186+
6B /r ib   imul r32, r/m32, imm8  80386+
6B /r ib   imul r16, imm8         80186+
6B /r ib   imul r32, imm8         80386+
69 /r iw   imul r16, r/m16, imm16 80186+
69 /r id   imul r32, r/m32, imm32 80386+
69 /r iw   imul r16, imm16        80186+
69 /r id   imul r32, imm32        80386+
```

Flags:

- Wenn `imul` nur einen Operanden hat, ist das Carry-Flag und das Overflow-Flag gesetzt, wenn ein Übertrag in die höherwertige Hälfte des Ergebnisses stattfindet. Passt das Ergebnis exakt in die untere Hälfte des Ergebnisses werden Carry- und Overflow-Flag gelöscht. Hat `imul` zwei oder drei Operanden, wird das Carry- und Overflow-Flag zu groß für den Zieloperanden, andernfalls wird es gelöscht
- SF, ZF, AF und PF sind undefiniert

9.10 INC (Increment)

Addiert 1 zum Zieloperanden.

```
FE /0      inc r/m8       8086+
FF /0      inc r/m16      8086+
40 r+w     inc r16        8086+
FF /6      inc r/m32      80386+
40 r+d     inc r32        80386+
```

Flags:

- OF, SF, ZF, AF, PF
- Das Carry-Flag wird nicht beeinflusst

Beispiel:

```
INC eax
INC [eax]
```

9.11 INT (Interrupt)

INT löst einen Software-Interrupt aus. Dies ist vergleichbar mit dem Aufruf eines Unterprogramms (hier ein Interrupt-Handler). Über die allgemeinen CPU-Register können dem Interrupt-Handler Werte übergeben werden. Der Interrupt-Handler kann über diese Register auch Werte zurückliefern. Nach INT steht eine Interrupt-Nummer, die zwischen 00h und FFh liegt. Über diese kann man die Lage der Einsprungadresse für den entsprechenden Interrupt-Handler in der Interrupt-Vektor-Tabelle ermitteln.

```
CD          int imm8        8086+
```

Flags:

(keine)

Beispiel:

```
MOV ah,4Ch;  Der Wert 4Ch (= Beenden des Programms) wird dem Interrupt-Handler
übergeben
INT 21h;     Der Interrupt-Handler wird aufgerufen. Die Interrupt-Nummer ist
21h (MS-DOS).
```

9.12 IRET (Interrupt Return)

Rückkehr aus dem Interrupt-Handler, welcher mit *INT* aufgerufen wurde. Dieser Befehl ist vergleichbar mit *RET,* jedoch werden hier die Flags restauriert.

```
CF          iret            8086+
```

Flags:

Alle Flag-Werte werden auf den Zustand vor dem Interrupt zurückgesetzt.

9.13 MOV (Move)

Mit dem MOV Befehl wird der zweite Operand in den ersten Operanden kopiert.

```
88 /r       mov r/m8, r8       8086+
89 /r       mov r/m16, r16     8086+
8A /r       mov r8, r/m8       8086+
8B /r       mov r16, r/m16     8086+
C6 /0       mov r/m8, imm8     8086+
C7 /0       mov r/m16, imm16   8086+
8C /r       mov r/m16, SReg    8086+
8E /r       mov SReg, r/m16    8086+
A0          mov al, moffs8     8086+
A1          mov ax, moffs16    8086+
A2          mov moffs8, al     8086+
A3          mov moffs16, ax    8086+
```

Flags:

(keine)

Beispiel:

```
mov eax,10
mov eax,ebx
```

9.14 MUL (unsigned Multiplication)

Multipliziert den Zieloperanden mit dem AL-, AX- oder EAX-Register. Das Ergebnis wird im AX-, DX:AX- oder EDX:EAX-Register abgelegt (abhängig von der Größe des Zieloperanden). Der höherwertige Anteil des Ergebnisses befindet sich im AH-, DX- bzw. EDX-Register.

```
F6 /4       mul r/m8           8086+
F7 /4       mul r/m16          8086+
F7 /4       mul r/m32          80386+
```

Flags:

- OF und CF werden auf 0 gesetzt wenn der höherwertige Anteil des Ergebnisses 0 ist

9.15 NEG (Two's Complement Negation)

Bildet das Zweierkomplement indem der Operand von 0 abgezogen wird.

```
F6/3        neg r/m8            8086+
F7/3        neg r/m16           8086+
F7/3        neg r/m32           80386+
```

Flags:

- OF, SF, ZF, AF und PF
- Setzt das Carry-Flag auf 0, wenn das Ergebnis 0 ist, andernfalls wird es auf 1 gesetzt.

9.16 NOP (No Operation)

Führt keine Aktion aus.

```
90          nop                 8086+
```

(Der Opcode entspricht `xchg ax,ax` bzw. `xchg eax,eax`.)

Flags:

(keine)

9.17 NOT

Dreht den Wahrheitswert der Bits um. Entspricht dem Einerkomplement.

```
F6 /2       not r/m8            8086+
F7 /2       not r/m16           8086+
F7 /2       not r/m32           80386+
```

Flags:

(keine)

9.18 OR

Verknüpft die Operanden durch ein logisches ODER.

```
0C ib       or AL, imm8         8086+
0D iw       or AX, imm16        8086+
80 /1 ib    or r/m8, imm8       8086+
```

```
81 /1 iw    or r/m16, imm16   8086+
83 /1 ib    or r/m16, imm8    8086+
08 /r       or r/m8, r8       8086+
09 /r       or r/m16, r16     8086+
0A /r       or r8, r/m8       8086+
0B /r       or r16, r/m16     8086+
0D id       or EAX, imm32     80386+
81 /1 id    or r/m32, imm32   80386+
09 /r       or r/m32, r32     80386+
83 /1 ib    or r/m32, imm8    80386+
0B /r       or r32, r/m32     80386+
```

Flags:

- OF und CF werden gelöscht
- SF, ZF und PF
- AF ist undefiniert

9.19 SBB (Subtraction with Borrow)

SBB subtrahiert von einem Speicherbereich oder einem Register den Wert eines Speicherbereichs oder eines Registers und berücksichtigt dabei das Carry-Flag.

```
1C ib       sbb AL, imm8       8086+
1D iw       sbb AX, imm16      8086+
80 /3 ib    sbb r/m8, imm8     8086+
81 /5 iw    sbb r/m16, imm16   8086+
83 /3 ib    sbb r/m16, imm8    8086+
18 /r       sbb r/m8, r8       8086+
19 /r       sbb r/m16, r16     8086+
1A /r       sbb r8, r/m8       8086+
1B /r       sbb r16, r/m16     8086+
1D id       sbb EAX, imm32     80386+
81 /3 id    sbb r/m32, imm32   80386+
83 /3 ib    sbb r/m32, imm8    80386+
19 /r       sbb r/m32, r32     80386+
1B /r       sbb r32, r/m32     80386+
```

Flags:

- OF, SF, ZF, AF, PF, CF

9.20 SHL (Shift Left)

Verschiebt alle Bits im Register um x Stellen nach links. Mathematisch wird der Wert im Register zum Beispiel beim Verschieben um eine Stelle mit 2 multipliziert. Beim Multiplizieren mit 2, 4 und so weiter sollte statt MUL besser dieser Befehl eingesetzt werden, da er von der CPU schneller abgearbeitet wird.

```
shl r8,x                    8086+
shl r16,x                   8086+
shl r32,x                  80386+
```

Flags:

- CF

Beispiel:

```
 mov al,00000001b (al = 00000001b)
shl al,1         (al = 00000010b)
```

9.21 SHR (Shift Right)

Verschiebt alle Bits im Register um x Stellen nach rechts. Mathematisch wird der Wert im Register zum Beispiel beim Verschieben um eine Stelle durch 2 dividiert.

```
shr r8,x                    8086+
shr r16,x                   8086+
shr r32,x                  80386+
```

Flags:

- ZF?

Beispiel:

```
 mov al,00000010b (al = 00000010b)
shr al,1         (al = 00000001b)
```

9.22 STC (Set Carry Flag)

Setzt das Carry-Flag.

```
F9          stc             8086+
```

Flags:

- CF

9.23 STI (Set Interrupt Flag)

Setzt das Interrupt-Flag.

```
FB          sti             8086+
```

Flags:

- IF

9.24 SUB (Subtract)

SUB subtrahiert von einem Speicherbereich oder einem Register den Wert eines Speicherbereichs oder eines Registers.

```
2C ib       sub AL, imm8        8086+
2D iw       sub AX, imm16       8086+
80 /5 ib    sub r/m8, imm8      8086+
81 /5 iw    sub r/m16, imm16    8086+
83 /5 ib    sub r/m16, imm8     8086+
28 /r       sub r/m8, r8        8086+
29 /r       sub r/m16, r16      8086+
2A /r       sub r8, r/m8        8086+
2B /r       sub r16, r/m16      8086+
2D id       sub EAX, imm32      80386+
81 /5 id    sub r/m32, imm32    80386+
83 /5 ib    sub r/m32, imm8     80386+
29 /r       sub r/m32, r32      80386+
2B /r       sub r32, r/m32      80386+
```

Flags:

- OF, SF, ZF, AF, PF, CF

Beispiel:

```
SUB eax,10
SUB eax,ebx
```

9.25 XCHG (Exchange)

Der erste und zweite Operand wird vertauscht.

```
90+rw       xchg ax, r16        8086+
90+rw       xchg r16, ax        8086+
86 /r       xchg r/m8, r8       8086+
```

```
86 /r       xchg r8, r/m8      8086+
87 /r       xchg r/m16, r16    8086+
87 /r       xchg r16, r/m16    8086+
```

Flags:

(keine)

9.26 XOR

Verknüpft die Operanden durch ein logisches exklusives Oder (Antivalenz).

```
34 ib       xor AL, imm8       8086+
35 iw       xor AX, imm16      8086+
80 /6 ib    xor r/m8, imm8     8086+
81 /6 iw    xor r/m16, imm16   8086+
83 /6 ib    xor r/m16, imm8    8086+
30 /r       xor r/m8, r8       8086+
31 /r       xor r/m16, r16     8086+
32 /r       xor r8, r/m8       8086+
33 /r       xor r16, r/m16     8086+
35 id       xor EAX, imm32     80386+
81 /6 id    xor r/m32, imm32   80386+
31 /r       xor r/m32, r32     80386+
83 /6 ib    xor r/m32, imm8    80386+
33 /r       xor r32, r/m32     80386+
```

Flags:

- OF und CF werden gelöscht
- SF, ZF und PF
- AF ist undefiniert

10 Befehlsliste

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
AAA	ASCII adjust for addition	AC	
AAD	ASCII adjust for division	PSZ	
AAM	ASCII adjust for multiply	PSZ	
AAS	ASCII adjust for subtraktion	AC	
ADC	Add with carry	ACOPSZ	Addiere zwei Operanden und CY
ADD	Addition	ACOPSZ	Addition ohne Übertrag
AND	And: logical conjunction	COPSZ	Logische Operation "UND"
CALL			Unterprogramm aufrufen
CBW	Convert byte to word		AH wird mit Bit 7 von AL gefüllt. Code 98H - 10011000
CLC	Clear carry flag	C	Lösche CF. Code F8H = 11111000
CLD	Clear direction flag		Lösche DF. Code FCH = 11111100
CLI	Clear interrupt enable flag		Sperrt maskierbare Interrupts. Code FAH = 11111010
CMC	Complement carry flag	C	Komplementiere CF. Code F5H = 11110101
CMP	Compare two operands	ACOPSZ	Logischer Vergleich: Die Operanden werden subtrahiert, die Flags gesetzt, das Ergebnis verworfen.

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
CMPSB[1]	Compare byte string	ACOPSZ	Das durch ES:[DI] adressierte Byte wird vom Operanden DS:[SI] subtrahiert, Flags gesetzt, das Ergebnis verworfen. SI und DI werden geändert: +1, wenn (DF)=0, sonst -1. Code 1010011w
CMPSW[2]	Compare word string	ACOPSZ	Das durch ES:[DI] adressierte Wort wird vom Operanden DS:[SI] subtrahiert, Flags gesetzt, das Ergebnis verworfen. SI und DI werden geändert: +2, wenn (DF)=0, sonst -2. Code 1010011w
CWD	Convert word to doubleword		Vorzeichengerechte Erweiterung: DX wird mit Bit 15 von AX gefüllt. Code 99H = 10011001
DAA	Decimal adjust for addition	ACPSZ	Korrigiere AL nach BCD-Addition
DAS	Decimal adjust for subtraction	ACPSZ	Korrigiere AL nach BCD-Subtraktion
DEC	Decrement destination by 1	AOPSZ	Operand wird um 1 verringert. Operand ist Register oder Speicher, Byte oder Wort.
DIV			Vorzeichenlose Division
ESC			Speicherzugriff für Coprozessoren

1 http://de.wikibooks.org/wiki/%2F_Beispiele%23CMPS
2 http://de.wikibooks.org/wiki/%2F_Beispiele%23CMPS

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
HLT			Die CPU hält an bis Reset oder einem erlaubten externen Interrupt. Code: F4H = 11110100
IDIV	Integer division, signed	ACOPSF	Vorzeichengerechte Integer-Division
IMUL	Integer multiply accumulator by register-or-memory; signed	CO	Integer-Multiplikation mit Vorzeichen
IN	Input byte / word		Überträgt Byte/Wort vom Eingabeport (Adr. im DX oder als Direktoperand < 255) nach AL/AX.
INC	Increment destination by 1	AOPSZ	Operand wird um 1 vergrößert.
INT	Interrupt		Software-Interrupt: Indirekter FAR-Sprung über Interruptvektor 0 ... 255 mit Speichern der Rückkehradresse.
INTO	Interrupt if overflow		Wenn OF wird INT 4 ausgeführt.
IRET	Interrupt return		Rücksprung aus INT- bzw. INTO- Routine.
JA	Jump if above		Sprung wenn Ergebnis größer ist
JAE	Jump if above or equal		Sprung wenn größer oder gleich
JB	Jump if below		Sprung wenn Ergebnis kleiner ist
JBE	Jump if below or equal		Sprung wenn Ergebnis kleiner/gleich
JC	Jump if Carry		Sprung wenn CF gesetzt ist
JCXZ	Jump if CX = 0		Sprung wenn Register CX = 0 ist
JE	Jump if equal		Sprung wenn log. Ergebnis gleich ist

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
JG			Sprung wenn Ergebnis einer log. Operation arithmetisch größer ist
JGE			Sprung wenn Ergebnis arithmetisch größer oder gleich ist
JL			Sprung wenn Ergebnis einer log. Operation arithmetisch kleiner ist
JMP	Jump		Springe zu angegebener Adresse
JNA	Jump if not above		Sprung wenn Ergebnis nicht größer ist
JNAE	Jump if not above or equal		Sprung wenn nicht größer oder gleich
JNB	Jump if not below		Sprung wenn Ergebnis nicht kleiner ist
JNBE	Jump if not below or equal		Sprung wenn Ergebnis nicht kleiner/gleich
JNC	Jump if not Carry		Sprung wenn CF gelöscht ist
JNE	Jump if not equal		Sprung wenn log. Ergebnis nicht gleich ist
JNG	Jump if not greater		Sprung wenn Ergebnis einer log. Operation nicht arith. größer ist
JNGE	Jump if not greater or equal		Sprung wenn Ergebnis nicht arithmetisch größer oder gleich ist
JNL			Sprung wenn Ergebnis einer log. Operation nicht arith. klein ist
JNLE			Sprung wenn Ergebnis einer log. Op. nicht arith. kleiner oder gleich ist
JNO			Sprung, wenn Flag OF nicht gesetzt ist

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
JNP			Sprung bei ungerader Parität (Flag PF gelöscht)
JNS	Jump if not signed		Sprung wenn Ergebnis einer log. Operation positiv ist
JNZ	Jump if non zero		Sprung, wenn log. Ergebnis ungleich ist.
JO			Sprung, wenn Flag OF gesetzt ist
JP			Sprung bei gerader Parität (Flag PF gesetzt)
JPO			Sprung bei ungerader Parität (Flag PF gelöscht)
JS	Jump if signed		Sprung wenn Ergebnis einer log. Operation negativ ist
JZ	Jump if zero		Sprung, wenn log. Ergebnis gleich ist.
LAHF	Load AH from flags	SZAPC	Bits von AH werden mit Flags gefüllt: Bit 7->S, 6->Z, 4->A, 2->P, 0->C. Code: 9FH = 10011111
LDS[3]	Load data segment register)		Von einem DWORT (Pointer, Vektor) wird LOW in ein 16 Bit Register (außer SS-Reg) und HIGH ins DS- Register geladen
LEA	Load effective adress		Die Offset- Adr. eines Speicheroperanden wird in ein Register geladen. Nur verwenden, wenn die EA zur Laufzeit berechnet werden muss, sonst MOV ..., OFFSET ... verwenden!

3 http://de.wikibooks.org/wiki/%2F_Beispiele%23LDS

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
LES	Load extra-segment register		Lade Physikalische Adr. nach ES:
LOCK			Sperre den Bus
LODSB	Load byte string		Überträgt Byte DS:[SI] nach AL. SI + bzw. - um 1 je nach DF. Code 1010110w
LODSW	Load word string		Überträgt Word DS:[SI] nach AX. SI + bzw. - um 2 je nach DF. Code 1010110w
LOOP	Iterate instruktion sequence until count complete		Das Count Register (CX) wird um 1 dekrementiert. Wenn CX #0, wird relativ gesprungen: IF (CX) #0 THEN (IP) := (IP) + disp(sign-extended to 16 bits. Code 11100010
LOOPZ	Loop on zero		identisch mit LOOPE
LOOPE	Loop on equal		CX := CX-1 . Solange CX #0 und wenn ZF =1 : Relativer Sprung. Code 11100001 disp
LOOPNZ	Loop on non zero		identisch mit LOOPNE
LOOPNE	Loop on not equal		CX := CX-1 . Solange CX #0 und wenn kein ZF: Relativer Sprung. Code 11100000 disp
MOV	MOVE		Lade Wert

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
MOVSB[4]	Move byte string		Speichert DS:[SI] nach ES:[DI], dann SI und DI + bzw. - um 1. Ist eine Kombination der Befehle LODSB und STOSB. Code 10100l0w
MOVSW[5]	Move word string		Speichert DS:[SI] nach ES:[DI], dann SI und DI + bzw. - um 2. Ist eine Kombination der Befehle LODSW und STOSW. Code 10100l0w
MUL	Multiply accumulator by register-or-memory; unsigned	CO	AL bzw. AX werden mit dem Operanden multipliziert. Obere Hälfte des Resultats in AH bzw. DX. Wenn High #0 ist, werden CF und OF gesetzt. Code 111101lw
NEG	Form 2's complement		Negation (Zweier-Komplement)
NOP	No operation		Code 90H = 10010000
NOT	Form 1's complement		invertiert Operand und speichert zurück.
OR	Or, inclusive	COPSZ	Zwei Operanden werden bitweise verknüpft, CF und OF gelöscht.
OUT	Output byte / word		siehe IN
POP	Pop word off stack into destination		Holt Wort aus Stack, speichert es nach Register oder Memory und erhöht SP um zwei.

4 http://de.wikibooks.org/wiki/%2F_Beispiele%23MOVS
5 http://de.wikibooks.org/wiki/%2F_Beispiele%23MOVS

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
POPF	Pop flags off stack	Alle	Hole Wort vom Stack ins Statusregister, Siehe PUSHF. Code 9DH = 10011101
PUSH	Push word onto stack		Wort in Stack speichern. Verringert SP um 2 und speichert Register bzw. Memory-Wort.
PUSHF	Push flags onto stack)		Rettet Flags in den Stack (siehe POPF) und verringert SP um 2.
RCL	Rotate left through carry	CO	Rotiere links durch das CF Flag
RCR	Rotate right through carry	CO	Rotiere rechts durch das Flag CF
REP[6]	Repeat string operation		FLAGS: Siehe konkrete String-Operation. Die nachfolgende primitive String- Operation (MOVS, SCAS, CMPS) wird wiederholt, wenn (CX) # 0 ist. CX wird um 1 verringert.
REPZ			identisch mit REPE
REPE[7]	Repeat string operation		FLAGS: Siehe konkrete String-Operation. Die nachfolgende primitive String- Operation (MOVS, SCAS, CMPS) wird wiederholt, wenn (CX) # 0 ist. CX wird um 1 verringert. Der Zyklus wird auch abgebrochen, wenn die Bedingung nicht erfüllt ist.

6 http://de.wikibooks.org/wiki/%2F_Beispiele%23REP
7 http://de.wikibooks.org/wiki/%2F_Beispiele%23REP

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
REPNE			identisch mit REPNZ
REPNZ[8]	Repeat string operation		FLAGS: Siehe konkrete String-Operation. Die nachfolgende primitive String- Operation (MOVS, SCAS, CMPS) wird wiederholt, wenn (CX) # 0 ist. CX wird um 1 verringert. Der Zyklus wird auch abgebrochen, wenn die Bedingung nicht erfüllt ist.
RET	Return from procedure		Lädt IP und evt. noch CS aus dem Stack und erhöht SP um 2 bzw. 4 (je nachdem ob Inter-Segment- bzw. Intra-Segment-Rücksprung. Wenn Konstant-Operand angegeben ist, wird er zum Stack addiert.
ROL[9]	Rotate left	CO	Rotiere nach links. Verschiebeanzahl im CL oder =1 ist möglich. Wenn Verschiebung nur einmalig, wird OF gesetzt, falls vor Verschiebung die 2 obersten Bits ungleich waren.
ROR	Rotate right	CO	Rotiere nach rechts. Siehe ROL

8 http://de.wikibooks.org/wiki/%2F_Beispiele%23REP
9 http://de.wikibooks.org/wiki/%2F_Beispiele%23ROL

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
SAHF	Store AH to flags		Flags werden mit den Bits von AH geladen: `Bit 7 6 5 4 3 2 1 0` `Flag SF ZF xx AF xx PF xx CF`(xx = undefin. Wert). Code: 9EH = 10011110
SAL[10]= SHL	Shift arithmetic left and shift logical left	COPSZ	In untere Bits werden Nullen eingeschoben. Das oberste Bit wird nach CF geschoben. Verschiebeanzahl im CL oder =1. Wenn Verschiebung einmalig, wird OF gesetzt, falls vor Verschiebung die 2 obersten Bits ungleich waren.
SAR[11]	Shift arithmetic right	COPSZ	Das oberste Bit wird gleich nach dem alten obersten Bit eingeschoben. Das unterste Bit wird nach CF geschoben. Verschiebeanzahl im CL oder =1. Wenn Verschiebeanzahl=1, wird OF gesetzt, falls vor Verschiebung die beiden obersten Bits ungleich waren.
SBB	Subtrakt with borrow	ACOPSZ	Vorzeichenlose Subtraktion inclusive CF

10 `http://de.wikibooks.org/wiki/%2F_Beispiele%23SAL`
11 `http://de.wikibooks.org/wiki/%2F_Beispiele%23SAR`

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
SCASB[12]	Scan byte string	ACOPSZ	Das Byte ES[DI] wird mit AL verglichen. Dann wird DI um 1 erhöht (wenn DF=0) bzw. erniedrigt.
SCASW[13]	Scan word string	ACOPSZ	Das Wort ES[DI] wird mit AX verglichen. Dann wird DI um 2 erhöht (wenn DF=0) bzw. erniedrigt.
SHL	Shift logical left		siehe bei SAL
SHR[14]	Shift logical right	COPSZ	Schiebe nach rechts. In obere Bits werden Nullen eingeschoben. Das unterste Bit wird nach CF geschoben. Verschiebeanzahl im CL oder =1. Wenn Verschiebeanzahl =1, wird OF gesetzt, falls vor Verschiebung die beiden obersten Bits ungleich waren.
STC	Set carry flag	C	Setze das Carry Flag. Code: F9H = 11111001
STD	Set direction flag	D	Setze das Zählrichtungs-Flag DF. Code: FDH = 11111101
STI	Set interrupt enable flag		Erlaubt maskierbare externe Interrupts nach Ausführung des nächsten Befehls. Code: FBH = 11111011

12 http://de.wikibooks.org/wiki/%2F_Beispiele%23SCAS
13 http://de.wikibooks.org/wiki/%2F_Beispiele%23SCAS
14 http://de.wikibooks.org/wiki/%2F_Beispiele%23SHR

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
STOSB	Store byte string		AL wird nach ES:[DI] gespeichert. DI wird um 1 erhöht, wenn DF =0 ist, sonst dekrementiert. Code: 101010lw
STOSW	Store word string		AX wird nach ES:[DI] gespeichert. DI wird um 2 erhöht, wenn DF =0 ist, sonst dekrementiert. Code: 101010lw
SUB	Subtrakt	ACOPSZ	Vorzeichenlose Subtraktion
TEST	Test, or logical compare	COPSZ	Operanden werden verglichen (AND). CF und OF werden gelöscht.
WAIT			Warten, bis der BUSY-Eingang nicht aktiv ist
XCHG	Exchange		Byte oder Wort wird zwischen Registern oder Reg. und Speicher ausgetauscht. Für Sreg nicht anwendbar.

Befehlsliste i8086			
Code	Bezeichnung	Flags	Beschreibung
XLAT[15]	Translate		Übersetzen: AL := <DS:BX+AL> Lädt ein Byte aus einer Tabelle (im DS) nach AL. Das AL- Reg. wird als Index einer durch das BX- Register adressierten Tabelle verwendet. Der so adressierte Operand wird nach AL geladen. Code: D7H = 11010111. z.B. MOV BX, OFFSET TAB_NAME, XLAT BYTE PTR [BX]
XOR	Exclusive or	COPSZ	Operanden werden verglichen und Bits gesetzt, wo ungleich. CF und OF werden gelöscht.

15 http://de.wikibooks.org/wiki/%2F_Beispiele%23XLAT

11 Literatur und Weblinks

11.1 Quellenangaben

[Nat00] *International System of Units (SI) – Prefixes for binary multiples* National Institut of Standard and Technology `http://physics.nist.gov/cuu/Units/binary.html`

`http://`

[Sie04] *Prozessorgrundlagen: Von-Neumann-Architektur, Teil* Prof. Dr. Christian Siemers. `http://www.tecchannel.de/technologie/prozessoren/402283/`

`http://`

[Pod03] *Das Assembler-Buch I – Grundlagen, Einführung und Hochsprachenoptimierung* ISBN 3827319293, Addison-Wesley, Deutschland, 2002

11.2 Weitere Weblinks

11.2.1 Assemblerprogrammierung

- Entwicklungsseite von MenuetOS, einem 32-Bit-Betriebssystem das komplett in Assembler entwickelt wird und unter der GPL steht[1]
- Sehr umfangreiches Buch über die Assemblerprogrammierung unter DOS, Linux und Windows. (englisch)[2]
- Kurze Einleitung zum Thema Assembler in Kombination mit Pascal.[3]

11.2.2 NASM

- Neben einer weiteren Downloadmöglichkeit für NASM auch ein Assembler Tutorial von Dr. Paul Carter (englisch)[4]

1 `http://www.menuetos.org/`
2 `http://webster.cs.ucr.edu/`
3 `http://assembler.hpfsc.de/`
4 `http://webster.cs.ucr.edu/AsmTools/NASM/`

- Neben einigen nützlichen Weblinks auch einige Beispielprogramme (englisch)[5]

11.2.3 FASM

- Homepage des Flat-Assemblers[6]

11.2.4 MMX/SSE

- Ausführliche Darstellung der MMX Technologie in der Zeitschrift C't 1/97[7]

11.2.5 Windows Programmierung

- Tutorial zur Windows Programmierung[8]
- Iczelions Win32Asm Tutorials[9]

11.2.6 Hardware

- Bericht über den Itanium Prozessor im Linux Magazin[10]
- Die Intelprozessoren im Überblick im Elektronik-Kompendium[11]
- "Moderne Prozessorarchitekturen" von der Uni Trier[12]
- Sehr umfassende technische Dokumentation zu x86 Prozessoren[13]

11.2.7 Source Code

- Assemblerprogramme für Windows und DOS[14]

11.2.8 Weitere Themen

- Geschichte von Intel und deren Prozessoren (von Intel)[15]

5 http://www.csee.umbc.edu/help/nasm/
6 http://www.flatassembler.com/
7 http://www.heise.de/ct/97/01/228/
8 http://www.deinmeister.de/wasmtut.htm
9 http://www.joachimrohde.com/cms/xoops/modules/articles/index.php?cat_id=2
10 http://www.linux-magazin.de/Artikel/ausgabe/2001/10/itanium/itanium.html
11 http://www.elektronik-kompendium.de/sites/com/0311051.htm
12 http://www.syssoft.uni-trier.de/systemsoftware/Download/Fruehere_Veranstaltungen/Seminare/Prozessorarchitekturen/Prozessorarchitekturen-1.html
13 http://www.sandpile.org/
14 http://www.asmsource.8k.com/
15 http://www.intel.com/corporate/pressroom/emea/deu/archive/mappen/die_intel_geschichte.htm

11.2.9 Interruptlisten

- *Ralf Brown's Interrupt List* HTML-Version. (englisch)[16]
- *Ralf Brown's Interrupt List* Download-Version. (englisch)[17]

11.3 Weitere Literatur

- Marcus Roming, Joachim Rohde: Assembler - Grundlagen der Programmierung, mitp-Verlag, 2003, ISBN 382660671X.
- Wolfgang Links, Assembler Programmierung; Franzis Verlag GmbH, 2004, ISBN 3772370144.
- Trutz Eyke Podschun: Das Assembler-Buch; Grundlagen und Hochsprachenoptimierung, Addison-Weslay, 1999, ISBN 3827315131.
- Peter Monadjemi: PC-Programmieren in Maschinensprache, Markt & Technik, ISBN 3-89090-957-4 (nicht mehr verfügbar).
- Don & Penn Brumm: 80386; Markt & Technik, 1995, ISBN 3890905919 (wahrscheinlich nicht mehr verfügbar).
- Robert Hummel: Die Intel-Familie – Technisches Referenzhandbuch für den 80x86 und 80x87; Ziff-Davis Press; 1995, ISBN 389362807X.
- Maurus, Reinhold; Wohak, Bertram: 80x86/Pentium Assembler; IWT, 1996 ISBN 382662601X.
- Podschun, Trutz Eyke: Die Assembler-Referenz II – Kodierung, Dekodierung und Referenz; Addison-Wesley, 2003, ISBN 3827320151.
- Osborne, Adam: Einführung in die Mikrocomputertechnik; TEWI, 1983, ISBN 3921803128.

16 `http://www.ctyme.com/intr/int.htm`
17 `http://www.cs.cmu.edu/~ralf/files.html`

12 Autoren

Edits	User
11	Appaloosa[1]
1	Berni[2]
159	Daniel B[3]
19	Dirk Huenniger[4]
24	Geitost[5]
8	Gronau[6]
1	Heuler06[7]
1	JackPotte[8]
1	JohannWalter[9]
2	Juetho[10]
43	Klaus Eifert[11]
1	Mrieken[12]
7	Pb[13]
13	Robert[14]
1	Sannaj[15]
2	StYxXx[16]
19	ThePacker[17]
1	Transporter[18]
2	Vernanimalcula[19]
3	Worker[20]

1 http://de.wikibooks.org/wiki/Benutzer:Appaloosa
2 http://de.wikibooks.org/wiki/Benutzer:Berni
3 http://de.wikibooks.org/wiki/Benutzer:Daniel_B
4 http://de.wikibooks.org/wiki/Benutzer:Dirk_Huenniger
5 http://de.wikibooks.org/wiki/Benutzer:Geitost
6 http://de.wikibooks.org/wiki/Benutzer:Gronau
7 http://de.wikibooks.org/wiki/Benutzer:Heuler06
8 http://de.wikibooks.org/wiki/Benutzer:JackPotte
9 http://de.wikibooks.org/wiki/Benutzer:JohannWalter
10 http://de.wikibooks.org/wiki/Benutzer:Juetho
11 http://de.wikibooks.org/wiki/Benutzer:Klaus_Eifert
12 http://de.wikibooks.org/wiki/Benutzer:Mrieken
13 http://de.wikibooks.org/wiki/Benutzer:Pb
14 http://de.wikibooks.org/wiki/Benutzer:Robert
15 http://de.wikibooks.org/wiki/Benutzer:Sannaj
16 http://de.wikibooks.org/wiki/Benutzer:StYxXx
17 http://de.wikibooks.org/wiki/Benutzer:ThePacker
18 http://de.wikibooks.org/wiki/Benutzer:Transporter
19 http://de.wikibooks.org/wiki/Benutzer:Vernanimalcula
20 http://de.wikibooks.org/wiki/Benutzer:Worker

Abbildungsverzeichnis

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- LGPL: GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. http://artlibre.org/licence/lal/de
- CFR: Copyright free use.

- EPL: Eclipse Public License. `http://www.eclipse.org/org/documents/epl-v10.php`

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[21]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

21 Kapitel 13 auf Seite 109

1	Daniel B, Dirk Huenniger	
2	Daniel B[22], Daniel B[23]	CC-BY-SA-3.0
3	Daniel B[24], Daniel B[25]	CC-BY-SA-3.0
4	Daniel B[26], Daniel B[27]	CC-BY-SA-3.0
5	Daniel B[28], Daniel B[29]	CC-BY-SA-3.0
6	Daniel B[30], Daniel B[31]	CC-BY-SA-3.0
7	Daniel B[32], Daniel B[33]	CC-BY-SA-3.0
8	Daniel B[34], Daniel B[35]	CC-BY-SA-3.0

22 http:////commons.wikimedia.org/w/index.php?title=User:Daniel_B&action=edit&redlink=1
23 http:///w/index.php?title=User:Daniel_B&action=edit&redlink=1
24 http:////commons.wikimedia.org/w/index.php?title=Benutzer:Daniel_B&action=edit&redlink=1
25 http:///w/index.php?title=Benutzer:Daniel_B&action=edit&redlink=1
26 http:////commons.wikimedia.org/w/index.php?title=Benutzer:Daniel_B&action=edit&redlink=1
27 http:///w/index.php?title=Benutzer:Daniel_B&action=edit&redlink=1
28 http:////commons.wikimedia.org/w/index.php?title=Benutzer:Daniel_B&action=edit&redlink=1
29 http:///w/index.php?title=Benutzer:Daniel_B&action=edit&redlink=1
30 http:////commons.wikimedia.org/w/index.php?title=Benutzer:Daniel_B&action=edit&redlink=1
31 http:///w/index.php?title=Benutzer:Daniel_B&action=edit&redlink=1
32 http:////commons.wikimedia.org/w/index.php?title=Benutzer:Daniel_B&action=edit&redlink=1
33 http:///w/index.php?title=Benutzer:Daniel_B&action=edit&redlink=1
34 http:////commons.wikimedia.org/w/index.php?title=Benutzer:Daniel_B&action=edit&redlink=1
35 http:///w/index.php?title=Benutzer:Daniel_B&action=edit&redlink=1

13 Licenses

13.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”. * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's “contributor version”.

A contributor's “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from

conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

13.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "freeïn the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Versionöf the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A SSecondary Sectionïs a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The Ïnvariant Sectionsäre certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Textsäre certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent"copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparentïs called Öpaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page"means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page"means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher"means any person or entity that distributes copies of the Document to the public.

A section Ëntitled XYZ"means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as Äcknowledgements", "Dedications", Ëndorsements", or "History".) To "Preserve the Titleöf such a section when you modify the Document means that it remains a section Ëntitled XYZäccording to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "Historyïn the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "Historyßection. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled Äcknowledgementsör "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled Ëndorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled Ëndorsementsör to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled Ëndorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "Historyïn the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled Äcknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled Ëndorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an ãggregateïf the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled Äcknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ör any later versionäpplies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site"(or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration"(or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA"means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

Ïncorporate"means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is ëligible for relicensingïf it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts."line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

13.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.