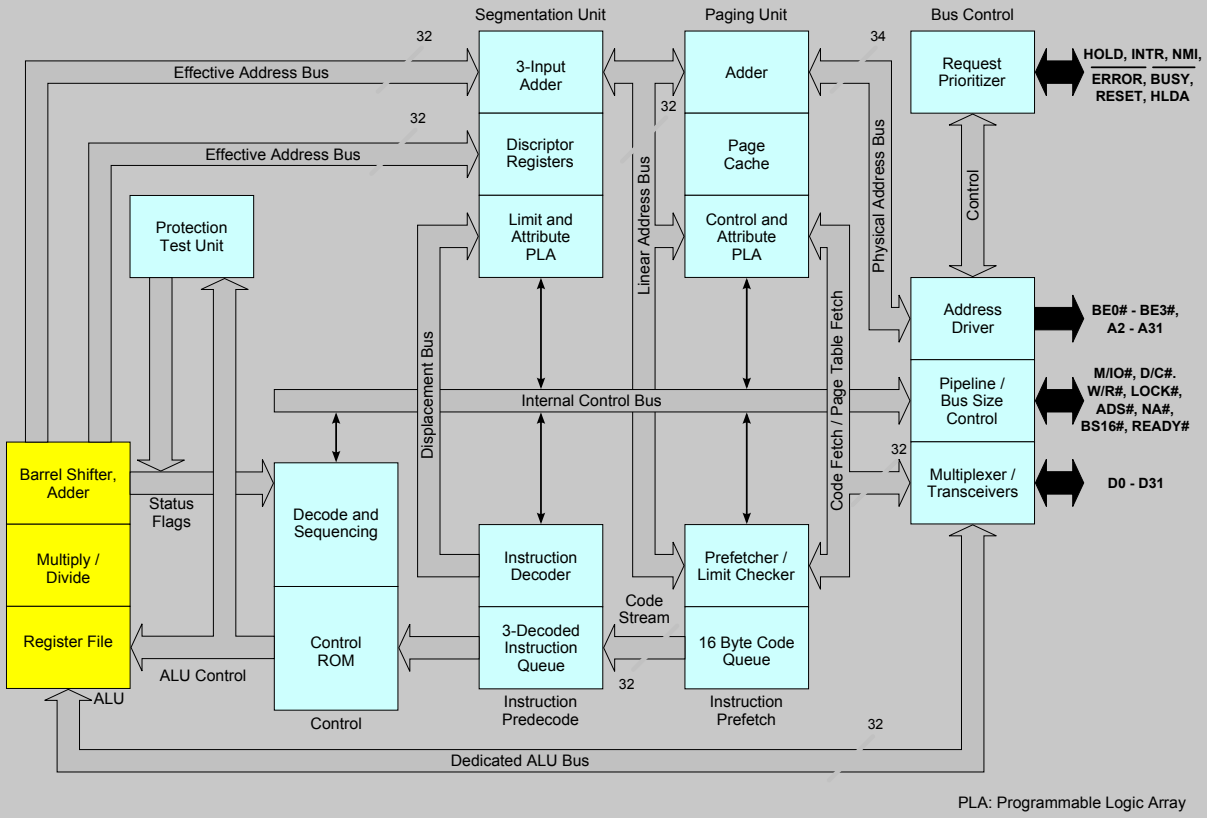
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Open C3S
Open Competence Center for Cyber Security



Zertifikatsprogramm

Kompaktkurs Intel-Assembler (IA-32) [MM-102]

Autoren:

Dr. rer. nat. Werner Massonne

Prof. Dr.-Ing. Felix C. Freiling

Friedrich-Alexander-Universität Erlangen-Nürnberg

Kompaktkurs Intel-Assembler (IA-32)
[MM-102]

Autoren:

Dr. rer. nat. Werner Massonne

Prof. Dr.-Ing. Felix C. Freiling

---

1. Auflage

Friedrich-Alexander-Universität Erlangen-Nürnberg

GEFÖRDERT VOM
Bundesministerium
für Bildung
und Forschung

© 2016 Felix Freiling
Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Martensstr. 3
91058 Erlangen

1. Auflage (9. Dezember 2016)

Das Werk einschließlich seiner Teile ist urheberrechtlich geschützt. Jede Verwendung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung der Verfasser unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Um die Lesbarkeit zu vereinfachen, wird auf die zusätzliche Formulierung der weiblichen Form bei Personenbezeichnungen verzichtet. Wir weisen deshalb darauf hin, dass die Verwendung der männlichen Form explizit als geschlechtsunabhängig verstanden werden soll.

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung, und Forschung unter dem Förderkennzeichen 16OH12022 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

Inhaltsverzeichnis

Einleitung **4**
I. Abkürzungen der Randsymbole und Farbkodierungen 4
II. Zu den Autoren 5
III. Lehrziele 6

Kompaktkurs Intel-Assembler (IA-32) **7**
1 Lernergebnisse 7
2 Die Prozessorarchitektur IA-32 7
3 Register 8
4 Instruktionssatz 13
4.1 Datentransferbefehle 16
4.2 Arithmetische und logische Befehle 16
4.3 Shift- und Rotationsbefehle 17
4.4 Kontrolltransferbefehle 18
4.5 Speicher und Adressierung 23
4.6 Subroutinen 27
4.7 Speicherverwaltung 35
4.8 String-Befehle 43
4.9 Sonstige Befehle 45
4.10 Interrupts und Exceptions 46
4.11 Maschinenbefehlsformat 47
5 Zusammenfassung 48
6 Übungen 50

Stichwörter **53**

Liste der Lösungen zu den Kontrollaufgaben **55**

Verzeichnisse **59**
I. Abbildungen 59
II. Exkurse 59
III. Tabellen 59
IV. Literatur 60

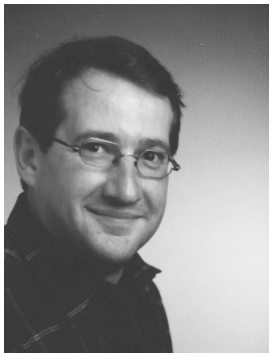
Einleitung

I. Abkürzungen der Randsymbole und Farbkodierungen

Exkurs	E
Übung	Ü

II. Zu den Autoren

Felix Freiling ist seit Dezember 2010 Inhaber des Lehrstuhls für IT-Sicherheitsinfrastrukturen an der Friedrich-Alexander-Universität Erlangen-Nürnberg. Zuvor war er bereits als Professor für Informatik an der RWTH Aachen (2003-2005) und der Universität Mannheim (2005-2010) tätig. Schwerpunkte seiner Arbeitsgruppe in Forschung und Lehre sind offensive Methoden der IT-Sicherheit, technische Aspekte der Cyberkriminalität sowie digitale Forensik. In den Verfahren zur Online-Durchsuchung und zur Vorratsdatenspeicherung vor dem Bundesverfassungsgericht diente Felix Freiling als sachverständige Auskunftsperson.



Werner Massonne erwarb sein Diplom in Informatik an der Universität des Saarlandes in Saarbrücken. Er promovierte anschließend im Bereich Rechnerarchitektur mit dem Thema „Leistung und Güte von Datenflussrechnern“. Nach einem längeren Abstecher in die freie Wirtschaft arbeitet er inzwischen als Postdoktorand bei Professor Freiling an der Friedrich-Alexander-Universität.

III. Lehrziele

Systemnahe Programmierung nimmt auf einem tiefen Level Bezug auf die Gegebenheiten eines Rechnersystems. Systemnahe Programmierung ist nicht mit Systemprogrammierung zu verwechseln. Systemprogrammierung benutzt die systemnahe Programmierung insbesondere zur Implementierung von Betriebssystemen oder ganz eng an Betriebssystem und Hardware angelehnter Softwarekomponenten wie Treiber, Prozesskommunikation usw.

Das wesentliche Merkmal der systemnahen Programmierung ist die direkte Kommunikation mit der Hardware und dem Betriebssystem eines Rechners. Diese erfordert elementare Kenntnisse bzgl. Rechnerarchitektur und Betriebssystemen im Allgemeinen. Solche Kenntnisse werden in diesem Mikromodul als bekannt vorausgesetzt. Umfangreiche Hochsprachen wie C++ oder Java abstrahieren von der Hardware eines Rechners, eine direkte Interaktion ist hier gerade nicht gewollt. Daher wird die systemnahe Programmierung meist in vergleichsweise minimalistischen Sprachen durchgeführt, entweder direkt in der Maschinensprache eines Rechners, die wir hier als Assembler bezeichnen, oder in C.

Wir beschäftigen uns in diesem Mikromodul mit einer ganz realen Rechnerarchitektur, auf der heute ein Großteil der Arbeitsplatzrechner aufbaut, nämlich mit IA-32. Diese wird detailliert vorgestellt, und die Assemblerprogrammierung dieser Architektur wird intensiv erlernt. Das hierbei erworbene Wissen kann relativ leicht auf andere Rechnerarchitekturen übertragen werden.

Kenntnisse in Assemblerprogrammierung sind in einem weiteren Bereich sehr wichtig, nämlich beim sogenannten „Reverse Engineering“. Reverse Engineering ist ein wichtiger Teilbereich der digitalen Forensik und beschäftigt sich mit der Analyse unbekannter Software. Die zu untersuchende Software liegt dabei in der Regel nur in Form von Maschinenprogrammen vor. Insbesondere bei Schad-Software (Malware) handelt es sich sehr häufig um IA-32-Maschinenprogramme. Um zu verstehen, was eine solche Schad-Software genau macht, muss man mit IA-32 und ihrer Programmierung vertraut sein.

Viel Spaß und viel Erfolg!

Kompaktkurs Intel-Assembler (IA-32)

1 Lernergebnisse

Sie können die Architektur IA-32 in die zeitliche Entwicklung der Intel-x86-Prozessorfamilie einordnen sowie charakterisieren. Sie haben aus Sicht der Assemblerprogrammierung grundlegende Kenntnisse über wesentliche Teile von IA-32 wie Register und Befehlssatz. Darüber hinaus können Sie die Besonderheiten wie Speicherverwaltung und Speicheradressierung dieser Architektur erläutern.

Sie haben einen umfassenden Überblick über IA-32 und können Assemblerprogramme für diese Architektur entwickeln und analysieren.

2 Die Prozessorarchitektur IA-32

In diesem Mikromodul beschäftigen wir uns mit der Intel x86-32-Bit-Architektur, die kurz als IA-32 (IA-32 steht für *Intel Architecture 32-Bit*) bezeichnet wird. x86 steht für eine ganze Prozessorfamilie der Firma Intel. Aber auch diverse andere Hersteller bauen oder bauten x86-kompatible Prozessoren wie z. B. AMD und Cyrix.

Der 8086-Prozessor aus dem Jahre 1978 war der erste x86-Prozessor (ein 16-Bit-Prozessor). Es folgte eine kontinuierliche Weiterentwicklung bis zum heutigen Tage, wobei besonders auf Abwärtskompatibilität geachtet wurde. Die Programme für die älteren Prozessoren sind also immer noch auf den Prozessoren der Weiterentwicklungen lauffähig.

Beim 80286-Prozessor wurde mit dem *Protected Mode* ein spezieller Zugriffsmodus auf den Hauptspeicher mit Zugriffsprivilegien eingeführt.

Intel 80386

Der Intel 80386war der erste Intel-Prozessor mit konsequenter 32-Bit-Architektur (32-Bit-ALU, 32-Bit-Register sowie 32-Bit breite Daten- und Adressbusse). Außerdem kam Paging hinzu, ein Speicherverwaltungssystem, auf das wir im Laufe dieses Mikromoduls noch näher eingehen werden. Alle Befehle und Register, die im 80386 vorhanden waren, sind auch bei den Intel x86-Prozessoren der modernsten Generation noch vorhanden. Im Rahmen dieses Moduls wird IA-32 behandelt, auf der die 80386-CPU basiert.

Intel Reference Manuals

Für viele Details, die über den Inhalt dieses Mikromoduls hinausgehen, sei bereits an dieser Stelle auf die Intel Reference Manuals (Intel [1987, 2012])[1] zum 80386 verwiesen. Wegen der Komplexität von IA-32 wird jeder noch so erfahrene Assembler-Programmierer ab und zu die Reference Manuals benutzen müssen.

Beim Intel 80486-Prozessor wurde eine FPU (*Floating Point Unit*) integriert, um mit Fließkommazahlen rechnen zu können.

Die weiteren Generationen von Intel x86-Prozessoren brachten diverse Erweiterungen der Architektur selbst sowie der Register- und der Befehlssätze mit sich. Diese Erweiterungen sind jedoch für uns nur von sekundärem Interesse, sodass diese hier nicht weiter betrachtet werden.

IA-32 ist bis heute eine der dominantesten Prozessorarchitekturen überhaupt, da sie die Welt der (Microsoft-)PCs beherrscht. Ein Großteil der heute auf PCs laufenden Software wurde für IA-32 entwickelt. Damit ist keineswegs nur nutzbringende

[1] Ersteres ist komplett online erhältlich, bspw. unter css.csail.mit.edu/6.858/2011/readings/i386.pdf

Software gemeint, sondern auch Schad-Software, sogenannte Malware. Um für Malware einen möglichst großen Kreis an Adressaten zu haben, macht es aus Sicht von deren Verfassern Sinn, sie für IA-32 zu entwickeln. Um Malware analysieren und verstehen zu können, ist im Umkehrschluss die Kenntnis von IA-32 besonders wichtig.

IA-32 lässt sich kurz wie folgt charakterisieren:

- 32-Bit-Architektur
- CISC (Complex Instruction Set Computer)
- relativ mächtige und komplizierte Speicherverwaltung

Abb. 1: Intel 80386-CPU

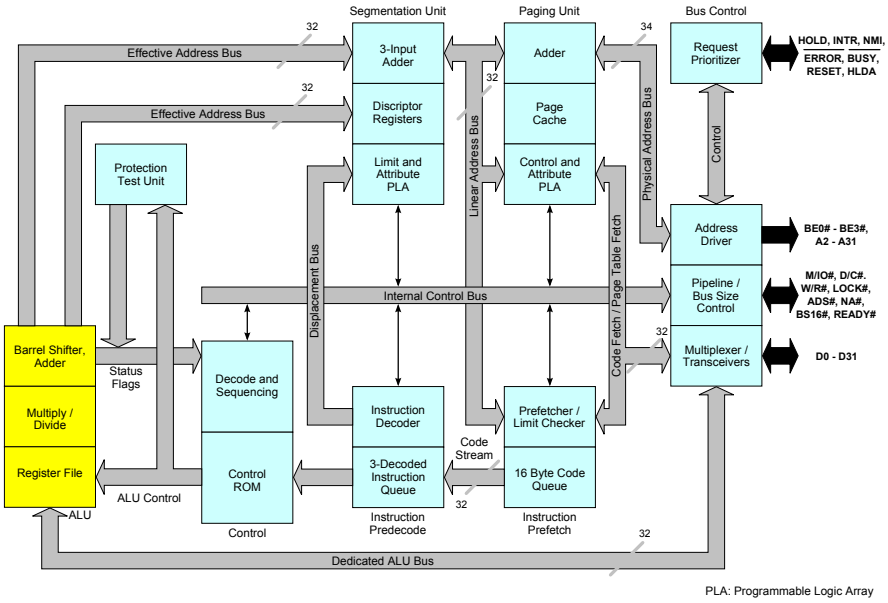


Abbildung 1[2] zeigt den Aufbau der Intel 80386-CPU. Aus Sicht des Programmierers sind die gelb eingezeichneten Bereiche wesentlich. Sie umfassen das eigentliche Rechenwerk (Addierer, Shifter, Multiplizierer usw.) und den Registersatz (*Register File*).

Darüber hinaus sind auf der rechten Seite die Einheiten zur Signalanbindung nach außen (Adressbus, Datenbus, Steuerleitungen), die *Bus Control Units* zu erkennen. Die Funktionalität zur Speicherverwaltung ist in der *Segmentation Unit* und der *Page Unit* implementiert.

Die Befehlsverarbeitung durchläuft nacheinander die *Instruction Prefetch Unit* mit einer Befehlswarteschlange (*Code Queue*), danach die *Instruction Predecode Unit* und schließlich die *Control Unit* mit der für CISC-Prozessoren typischen Sequentialisierung von komplexen Befehlen durch Mikrocode. Die *Protection Test Unit* überwacht schließlich die Zugriffsprivilegien des Protected Mode.

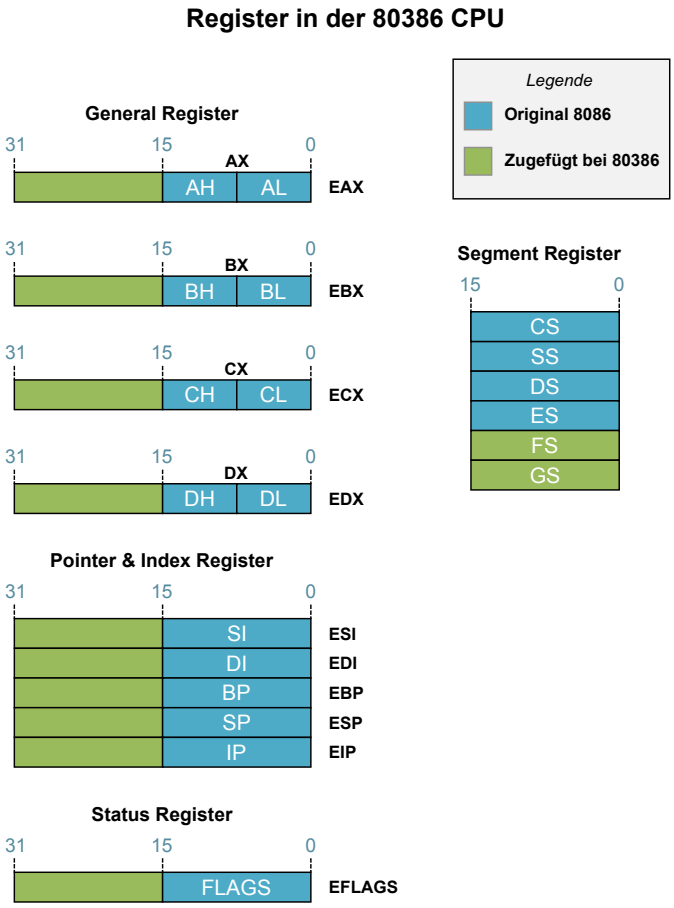
3 Register

Abb. 2[3] zeigt die Register der 80386-CPU. Die blau eingezeichneten Register sind die der 8086-CPU, die grünen Teile sind die Erweiterungen des 80386. Im Bereich

[2] Quelle: CC BY-SA 3.0, https://commons.wikimedia.org
[3] Quelle: www.tech-juice.org

der Segmentregister kamen zwei neue dazu, die übrigen Register wurden von 16 auf 32 Bit erweitert.

Abb. 2: Registersatz des 80386



Die Erweiterung der Register hat aus Kompatibilitätsgründen zur Folge, dass auch lediglich Teilbereiche eines Registers in einem 80386-Befehl angesprochen und manipuliert werden können. Abb. 3 zeigt das am Beispiel des Universalregisters `eax`. In Assembler wird das komplette 32-Bit-Register unter dem symbolischen Namen `eax` angesprochen. Die unteren 16 Bit sind unter dem (historischen) Namen `ax` separat anwählbar. Dieses 16-Bit-Register `ax` ist wiederum in zwei 8-Bit-Register `ah` und `al` unterteilt. Beispielsweise manipuliert ein Additionsbefehl auf Register `ah` nur die Bits 8 bis 15 von `eax`, ihre Modifikation hat keinen Einfluss auf die übrigen Bits von `eax`.

Abb. 3: Registerauf- teilung beim 80386

Die Register von IA-32 unterteilen sich in die folgenden Kategorien:

- Datenregister
- Indexregister
- Segment- bzw. Adressregister
- Stack-Register

- Systemregister

Daten- & Indexregister

Daten- und Indexregister werden zusammen als *General-Purpose*-Register bezeichnet, d. h. die Verwendung dieser Register ist weitgehend beliebig und dem Programmierer überlassen. Einige Befehle erwarten jedoch, dass Operanden in spezielle Register geladen werden. In Tabelle 1 sind die General-Purpose-Register des 80386 und jeweils typische Verwendungen dieser Register aufgeführt.

Tabelle 1: General Purpose Register

Registername	Typischer Verwendungszweck
eax	Zwischenwerte bei Berechnungen, Rückgabewerte
ebx	Basisadresse für Adressierungen
ecx	Zählregister in Schleifen, Index
edx	I/O–Daten, Double-Precision–Operationen
esi	Speicheradresse von String–Quelloperand
edi	Speicheradresse von String–Zieloperand

Segmentregister

Die Segmentregister der 8086-CPU sind für die Bildung der physikalischen Speicheradressen zuständig und 16 Bit breit. Diese sind das `cs`-Register (Codesegment), das `ds`-Register (Datensegment), das `es`-Register (Extrasegment für Daten) und das `ss`-Register (Stack-Segment). Beim 80386 wurden zwei weitere Extrasegmentregister `fs` und `gs` eingeführt.

Mit der Einführung des Protected Mode beim 80286 bekamen die Segmentregister eine andere logische Bedeutung. Die Segmentregister dienen allgemein zur Aufteilung des Hauptspeichers in unterschiedliche Abschnitte für Code und Daten. Details dazu werden im Abschnitt 4.7 angesprochen. Segmentierung hat aktuell keine große Bedeutung mehr (sie wurde durch *Paging* abgelöst). Zum Verstehen älterer Software muss man allerdings immer noch wissen, wie die Segmentierung sowohl beim 8086 als auch im Protected Mode funktioniert.

Stack- & Systemregister

Stack-Register und Systemregister bilden zusammen die *Special-Purpose*-Register. In Tab. 2 sind diese mit ihrer jeweilige Funktion aufgelistet.

Tabelle 2: Special Purpose Register

Registername	Verwendungszweck
esp	zeigt auf das „oberste“ Element des Stack („wächst“ nach unten)
ebp	zeigt auf den aktuellen Stack Frame
eip	zeigt auf die aktuell ausgeführte Instruktion
eflags	enthält diverse Bit-Flags; diese werden insbesondere für bedingte Sprünge verwendet

Stack-Register dienen der Verwaltung von Stack-Speicheradressen. Ein *Stack* (auch Stapel oder Keller genannt) ist ein reservierter Bereich im Hauptspeicher. Der Zugriff auf den Stack erfolgt nach dem LIFO-Prinzip (*Last In First Out*): Das zuletzt auf dem Stack abgelegte Datum[4] wird in der Regel wieder als erstes vom Stack genommen.

Der Stack ist für die Assemblerprogrammierung unter anderem für das Zwischenspeichern von Rücksprungadressen bei Unterprogrammaufrufen und zum Datenaustausch zwischen aufrufendem und aufgerufenem Programm wichtig. Details werden in Abs. 4.6 behandelt.

- Das `esp`-Register (*Stack Pointer*) enthält die Adresse des obersten Elements auf dem Stack. Man sagt allgemein auch, `esp` *zeigt* auf das oberste Element.

[4] „Datum“ ist hier der Singular von „Daten“.

- Das `ebp`-Register (*Base Pointer*) wird verwendet, um innerhalb des Stack auf gespeicherte Daten zuzugreifen. `ebp` zeigt auf den Anfang des aktuellen *Stack Frame*. Innerhalb des Hauptprogramms zeigt `ebp` auf den Anfang des Stack. Bei jedem Unterprogrammaufruf wird ein Stack Frame angelegt; das Unterprogramm erhält – vereinfacht ausgedrückt – seinen eigenen Abschnitt im Stack, auf dessen Anfang der Base Pointer zeigt. Stack Frames werden im Verlauf dieses Mikromoduls ausführlich behandelt.
- Das `eip`-Register (*Instruction Pointer*) enthält die Adresse des aktuell ausgeführten Maschinenbefehls.
- Das `eflags`-Register gibt Auskunft über den Prozessorstatus. Es ist ein spezielles „Überregister“, das im Prinzip, wie man der Abbildung 4 entnehmen kann, aus mehreren weitgehend unabhängigen Bits besteht. Jedes einzelne dieser Bits, die auch als *Flags* bezeichnet werden, geben einen bestimmten Zustand an. Die Flags können die Werte 0 und 1 annehmen. Man spricht auch von gelöschten und gesetzten Flags.

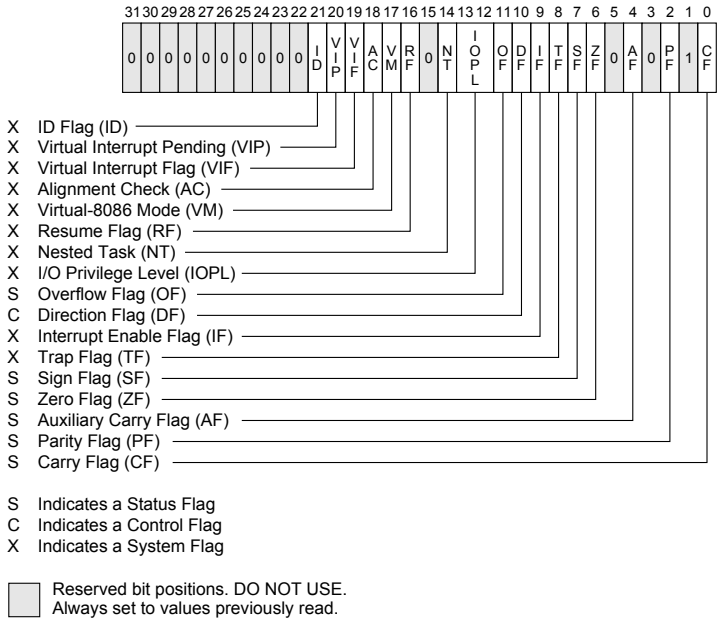


Abb. 4: Flags der 80386-CPU (Intel [1987])

Das Flag-Register ist eines der wenigen Register, die von Prozessorgeneration zu Prozessorgeneration erweitert wurden. Ursprünglich war das Flag-Register 16 Bit breit und nur neun dieser Bits hatten eine Bedeutung. Mit dem 80386 wurde das Register auf 32 Bit erweitert, wovon viele erst bei späteren Prozessoren der x86-Familie benutzt wurden.

Man unterscheidet Status-Flags und System-Flags.

Status-Flag

Status-Flags sind Flags, die der Prozessor nach arithmetischen oder bitweise logischen Operationen setzt, um etwas über das Resultat dieser Operation auszusagen. Die Status-Flags werden also in der Regel indirekt verändert. In den Reference Manuals zu IA-32 wird zu jeder Instruktion angegeben, welche Flags durch sie beeinflusst werden. Der Programmierer kann diese Flags dann in bedingten Sprungbefehlen abfragen und Programmverzweigungen vom Zustand der Flags abhängig machen.

Die wichtigsten Status-Flags sind:

Zeroflag: Das Zeroflag (ZF) wird gesetzt, wenn das Ergebnis der letzten arithmetischen oder bitweise logischen Operation Null war. Ansonsten wird es gelöscht.

Signflag: Das Signflag (SF) wird gesetzt, wenn das Ergebnis der letzten Operation negativ war. Ansonsten wird es gelöscht.

Carryflag: Das Carryflag (CF) wird gesetzt, wenn bei der letzten Operation der vorzeichenlose Wertebereich überschritten wurde. Ansonsten wird es gelöscht. Oder anders ausgedrückt: Das Carryflag wird gesetzt, wenn die Anzahl der vorhandenen Bits für das Ergebnis nicht ausreicht (siehe auch Overflowflag).

Overflowflag: Das Overflowflag (OF) wird immer durch das XOR (exklusives Oder) aus dem Übertrag, der bei der letzten Operation in das höchstwertige Bit hineingeht und dem Übertrag, der aus dem höchstwertigen Bit hinausgeht, berechnet. Es hat allerdings nur eine Bedeutung, wenn bei der Operation vorzeichenbehaftete Werte (Zweierkomplement) verwendet werden, dann zeigt es einen Überlauf des Wertebereichs an. Bei Berechnungen mit vorzeichenlosen Werten hat das Overflowflag keine Bedeutung.

Auxiliaryflag: Das Auxiliaryflag (AF) wird gesetzt, wenn ein Übertrag von Bit 3 auf Bit 4 stattfindet. Ansonsten wird es gelöscht. Das Flag ist hauptsächlich bei der Bearbeitung von BCD(*Binary Coded Decimal*)-Zahlen[5] von Bedeutung.

Parityflag: Das Parityflag (PF) wird gesetzt, wenn das Ergebnis nach einer Operation eine gerade Anzahl von `1`en im niederwertigsten Byte enthält. Ansonsten wird es gelöscht.

E

Exkurs 1: **System-Flags**

Zusätzlich zu den Status-Flags gibt es eine Reihe vonSystem-Flags, die einen bestimmten Systemstatus anzeigen. Von diesen sollen einige an dieser Stelle der Vollständigkeit wegen vorgestellt werden, auch wenn deren genaue Bedeutung teilweise erst in späteren Abschnitten dieses Mikromoduls klar werden wird.

Interruptflag: Ist das Interruptflag (IF) gelöscht, so werden keine weiteren Interrupts mehr bearbeitet, bis es wieder gesetzt wird. Das Interruptflag ist vor allem bei zeitkritischem Code wichtig.

Trapflag: Das Trapflag (TF) ist vor allem für Debugger (Fehlersuche + Programmanalyse) von Bedeutung. Ist das Trapflag (Einzelschritt-Flag) gesetzt, so wird nach jedem Befehl Interrupt 1 aufgerufen, was eine schrittweise Abarbeitung der Befehle ermöglicht.

Directionflag: Das Directionflag (DF) gibt bei String-Operationen vor, in welcher Richtung Strings bearbeitet werden sollen. Bei gelöschtem Flag werden die Strings nach aufsteigenden Adressen bearbeitet (`esi` und `edi` werden inkrementiert) und bei gesetztem Flag nach absteigenden Adressen (`esi` und `edi` werden dekrementiert).

I/O Privilege Level Field Flag (2 Bits): Das IOPL-Flag wurde erst mit dem 80286 eingeführt. Es ist nur im Protected Mode von Bedeutung. Es gibt an, welche Privilegstufe das aktuell laufende Programm haben muss, um direkte I/O-Operationen ausführen zu können. Ist die Privilegstufe des aktuellen Programms, auch CPL (*Current Privilege*

[5] In der BCD-Form wird jede dezimale Ziffer (0 bis 9) durch jeweils 4 Bits dargestellt.

Level) genannt, kleiner oder gleich dem IOPL, dann darf es direkt auf den I/O-Adressraum zugreifen. Das Flag kann nur durch die Befehle `popf` und `iret` modifiziert werden, wenn der aktuelle CPL gleich 0 ist.

Resumeflag: Das Resumeflag (RF) wurde erst mit dem 80386 eingeführt. Ist das Flag gesetzt, so werden mehrfache Debugging Exceptions unterdrückt, wenn ein Breakpoint auf einen bestimmten Befehl gesetzt wird. Da der Breakpoint vor Ausführung des Befehls erkannt und die Exception aufgerufen wird, ohne den Breakpoint vorher zu entfernen, wird nach Rückkehr von der Exception erneut derselbe Breakpoint erkannt, anstatt mit dem Befehl fortzufahren, auf den der Breakpoint gesetzt wurde.

Virtual 8086 Mode Flag: Das VM-Flag wurde erst mit dem 80386 eingeführt. Wird dieses Flag gesetzt, schaltet der Prozessor vom Protected Mode in den virtuellen 8086-Modus um, was ein Ausführen von normalen 8086-Programmen ermöglicht. Wird es gelöscht, wird wieder in den Protected Mode geschaltet.

Neben den bisher vorgestellten Registern verfügen der 80386-Prozessor und seine Nachfolger über viele weitere Register. Dies sind einmal Datenregister, insbesondere solche für die integrierten Fließkommaerweiterungen ab der 80486-CPU. Hierbei sind insbesondere die MMX(64 Bit)- und SSE(128 Bit)-Register zu nennen. Diese Register werden im Rahmen dieses Moduls nicht weiter betrachtet. Zum einen ist die technische Weiterentwicklung gerade hier so groß, dass eine aktuelle Bestandsaufnahme rasch veraltern würde. Zum anderen sind detaillierte Kenntnisse dieser Register und der damit verbundenen Befehlssatzerweiterungen für die grundsätzlichen Techniken der Assemblerprogrammierung und der Programmanalyse nicht erforderlich. Im Bedarfsfall wird es kein großes Problem darstellen, den Umgang mit diesen Erweiterungen selbstständig zu erlernen. Neben diesen Datenregistern existieren sehr viele weitere Steuerungsregister, von denen allerdings ein Großteil für uns nicht relevant ist. Die übrigen - wie zum Beispiel die für die Speicherverwaltung (Paging) - werden in den entsprechenden Abschnitten erklärt.

4 Instruktionssatz

IA-32 verfügt über einen – wie bei CISC-Architekturen üblich – umfangreichen Instruktionssatz.[6] In der Regel wird allerdings nur ein Teil des kompletten Instruktionssatzes verwendet, insbesondere Compiler beschränken sich oft auf nur wenige verschiedene Instruktionen. Grob kann der Instruktionssatz in folgende Klassen unterteilt werden, die wir schon prinzipiell im vorangegangenen Abschnitt kennengelernt haben:

- Datentransferbefehle (Lese-/Schreiboperationen zwischen Registern bzw. Speicher und Register)
- Arithmetische, logische und bitweise Befehle (Addition, AND, Shift, usw.)
- Kontrolltransferbefehle (statischer Sprung, bedingter Sprung)
- Subroutinenbefehle (Call, Return, usw.)
- String-Befehle (String kopieren, String-Vergleich, usw.)

[6] In diesem Modul werden die Begriffe „Instruktionssatz“ und „Befehlssatz“ sowie „Instruktion“ und „Befehl“ als Synonyme verwendet.

- sonstige Befehle wie z. B. I/O-Befehle und Flag-Kontrollbefehle

Diese Befehlsklassen werden im Folgenden detailliert vorgestellt und erkärt. Es werden allerdings jeweils nicht alle Befehle des sehr umfangreichen Befehlssatzes von IA-32 aufgelistet. Wir beschränken uns auf die für dieses Modul wichtigsten und relevantesten. Der vollständige Befehlssatz ist im Reference Manual zu finden.

Die generelle Befehlsdarstellung sieht in der *Intel-Syntax* wie folgt aus:

```
MN [Op1 [,Op2]]
```

`MN` ist das Befehls-Mnemonic, `Op1` und `Op2` sind optionale Operanden, also z. B. Register, Speicheradressen usw.

IA-32 ist eine 2-Register-Architektur

IA-32 oder allgemein die Architekturen der x86-Prozessorfamilie sind 2-Register-Architekturen. Das heißt, dass in einem Befehl maximal zwei unterschiedliche Register angesprochen werden können. Bei Operationen mit zwei Operanden (z. B. Addition) ist immer eines der beiden Register Zieloperand. Es wird mit dem Ergebnis der Addition überschrieben. In der Intel-Syntax ist immer der linke Operand der Zieloperand.

B

Beispiel 1

```
add eax, ecx
```

Dieser Assemblerbefehl bedeutet, dass der Inhalt von `eax` und `ecx` addiert und das Ergebnis in `eax` gespeichert wird.

Die Anweisung ist also gleichbedeutend mit:

`eax = eax+ecx`

Statt Register können die Operanden Hauptspeicherinhalte sein. Allerdings gilt die Einschränkung, dass innerhalb eines Befehls in der Regel nur ein Hauptspeicherzugriff erfolgen darf. Die Adressierung des Hauptspeichers (effektive Adresse) wird durch eckige Klammern dargestellt.

B

Beispiel 2

`add eax, [1234]` ist ein gültiger Befehl. `eax` und der Speicherinhalt an Adresse `1234` werden addiert, und das Ergebnis wird in `eax` abgelegt.

`add [1234], eax` ist **kein** gültiger Befehl. Der Speicherinhalt an Adresse `1234` müsste gelesen und das Ergebnis wieder in den Speicher transferiert werden. Das wären jedoch zwei Speicherzugriffe.

Bei manchen Befehlen ist es allerdings doch erlaubt, zuerst lesend und dann schreibend auf ein und dieselbe Speicherzelle zuzugreifen. Dies hängt von der Adressierungsart (s. Abs. 4.5) des Hauptspeichers ab. Beispielsweise ist der Befehl `add`

`[eax], ecx` gültig, bei dem die Adressierung indirekt über `eax` erfolgt. Die Gründe dafür liegen tief im Inneren der Architektur, und die Konsequenzen daraus erscheinen aus der Sicht eines Programmierers nicht immer logisch zu sein.

E

Exkurs 2: **Intel-Syntax vs. AT&T-Syntax**

Im Rahmen dieses Moduls wird die Intel-Syntax verwendet.

Die wichtigsten Konventionen dabei sind:

1. Der Zieloperand steht immer links.
2. Die Zahlendarstellung ist dezimal. Die hexadezimale Darstellung wird explizit mit einem "h" am Ende gekennzeichnet (z. B. 10h).
3. Die Operandengröße wird bei Speicherzugriffen nicht explizit angegeben, falls sie sich aus dem jeweiligen Kontext ergibt.
   Beispiel: `mov eax, [1234]` lädt einen 32-Bit-Operanden aus dem Speicher, da `eax` ein 32-Bit-Register ist.
4. Die effektive Adresse bei Speicherzugriffen steht in eckigen Klammern, d. h. die Berechnung des Ausdrucks in den eckigen Klammern ergibt die Adresse im Speicher.

Neben der Intel-Syntax ist die AT&T-Syntax gebräuchlich, z. B. beim GNU-Assembler. Die wichtigsten Konventionen der AT&T-Syntax sind:

1. Der Zieloperand steht immer rechts.
2. Konstanten werden mit $ gekennzeichnet, Register mit einem %.
3. Mnemonics enthalten die Operandengröße als *Suffix* (l = dword (32 Bit), w = word (16 Bit), b = byte (8 Bit)).
4. Die effektive Adresse bei Speicherzugriffen wird in der Schreibweise `DISP(BASE,INDEX,SCALE)` angegeben.
   (Die Speicheradressierung wird in Abs. 4.5 im Detail behandelt.)

Abb. 5 zeigt die Syntaxunterschiede an einem Programmbeispiel.

Intel Syntax	AT&T Syntax
`push_loop:`	`push_loop:`
`mov [ebp - 4], ecx`	`movl %ecx, -4(%ebp)`
`mov edx, [ebp + 8]`	`movl 8(%ebp), %edx`
`xor eax, eax`	`xorl %eax, %eax`
`mov al, byte [ebx + esi]`	`movb (%ebx, %esi, 1), %al`
`push eax`	`pushl %eax`
`push edx`	`pushl %edx`
`call printf`	`call printf`
`add esp, 8`	`addl $8, %esp`
`mov ecx, [ebp - 4]`	`movl -4(%ebp), %ecx`
`inc esi`	`incl %esi`
`loop push_loop`	`loop push_loop`
`push newline`	`pushl $newline`
`call printf`	`call printf`
`add esp, 4`	`addl $4, %esp`
`mov esp, ebp`	`movl %ebp, %esp`
`pop ebp`	`popl %ebp`
`ret`	`ret`

Abb. 5: Intel- und AT&T-Syntax

4.1 Datentransferbefehle

IA-32 verfügt über die beiden reinen Datentransferbefehle `mov` und `xchg`. Tab. 3 zeigt die Syntax dieser Befehle, ihre Bedeutung sowie Beispiele. Nach diesem Schema sind auch die weiteren Tabellen zum Befehlssatz aufgebaut.

Tabelle 3: Datentransferbefehle

Befehl	Operation	Beispiel
mov dest, source	dest=source	mov eax, ecx
		mov edx, [1234]
xchg op1, op2	op1=op2, op2=op1	xchg eax, edx
movzx op1, op2		movzx eax, cx
movsx op1, op2		movsx eax, cx

Der Befehl `mov` verschiebt ein Datum von der Quelle (*Source*) zum Ziel (*Destination*). `mov` ist mit den verschiedenen Adressierungsarten (s. Abs. 4.5) kombinierbar, allerdings darf nur einer der beiden Operanden den Hauptspeicher referenzieren. Der Befehl `xchg` vertauscht die Inhalte zweier Register. Datentransfers sind natürlich auch indirekt über die arithmetischen und logischen Befehle möglich.

Die Tabelle zeigt zwei weitere, spezielle Datentransferbefehle: `movzx` und `movsx` sind Typumwandlungsbefehle. Hierbei ist `op2` ein 8- oder 16-Bit-Register bzw. Speichervariable und `op1` ein 16- oder 32-Bit-Register. `op2` wird expandiert und nach `op1` verschoben. Ist `op2` kürzer als `op1`, so wird `op2` auf die passende Größe gebracht; hierbei werden bei `movzx` führende Nullen ergänzt, bei `movsx` wird das Vorzeichen expandiert.

B

Beispiel 3

Expansion des 8-Bit-Registers `bl` in das 16-Bit-Register `ax` ohne und mit Berücksichtigung des Vorzeichens:

```
mov     bl, -5  ; bl = fbh
movzx   ax, bl  ; ax = 00fbh

mov     bl, -5  ; bl = fbh
movsx   ax, bl  ; ax = fffbh = -5
```

4.2 Arithmetische und logische Befehle

Tabelle 4[7] zeigt die wichtigsten arithmetischen und logischen Befehle von IA-32. Die Funktionsweise sollte bis auf die Multiplikations- und Divisionsbefehle eindeutig sein. Zunächst (8086-Prozessor) existierten nur die Befehle `mul` und `div` für vorzeichenlose Operanden mit fest vorgegebenen Registern. Diese wurden um die Befehle `imul` und `idiv` für vorzeichenbehaftete Operanden ergänzt. Der `imul`-Befehl wurde zudem gegenüber `mul` stark erweitert. `imul` mit einem Operanden ist funktionsgleich zu `mul`. Bei `imul` mit zwei Operanden werden beide Operanden multipliziert; der linke Operand ist ein Register, das auch das Ergebnis der Multiplikation zugewiesen bekommt; der rechte Operand ist ein Register, eine Speicheradresse oder eine Konstante. Bei `imul` mit drei Operanden werden der zweite und dritte Operand multipliziert und das Ergebnis im ersten Operanden abgelegt. Der erste Operand ist hierbei ein Register, der zweite ein Register oder eine Speicheradresse und der dritte eine Konstante. Bei den Varianten mit zwei bzw. drei Operanden findet im Falle eines zu kleinen Ergebnisregisters kein Übertrag in ein anderes Register statt.

[7] Die Schreibweise `edx:eax` bedeutet, dass die beiden 32-Bit-Register `edx` und `eax` logisch zu einem 64-Bit-Register zusammengefasst sind.

Die unterschiedlichen Auswirkungen dieser Befehlsklasse auf das Register `eflags` können Sie im Bedarfsfall im Reference Manual nachlesen. Meist sind sie jedoch intuitiv klar, und die Spezialfälle sind für das weitere Verständnis nicht wichtig.

Tabelle 4: Arithmetische und logische Befehle

Befehl	Operation	Beispiel
add dest, source	dest=dest+source	add eax, ecx
sub dest, source	dest=dest-source	sub eax, ecx
adc dest, source	dest=dest+source+CF	adc eax, ecx Addition mit Übertrag
sbb dest, source	dest=dest-(source+CF)	sbb eax, ecx Subtraktion mit Übertrag
inc op	op=op+1	inc eax Inkrementierung um 1
dec op	op=op-1	inc eax Dekrementierung um 1
mul op	edx:eax=eax*op	mul ebx (obere 32 Bit des Ergebnisses stehen in edx)
div op	eax=edx:eax/op edx=Rest	div ebx eax = (edx:eax) *div* ebx edx = eax *mod* ebx Ganzzahlige Division mit ganzzahligem Rest
imul op1, [op2, [op3]]	s. Text	imul ebx, ecx
idiv op	wie div vorzeichenbehaftet	
and dest, source	dest=dest AND source	and eax, ecx Bitweise AND-Verknüpfung
or dest, source	dest=dest OR source	or eax, ecx Bitweise OR-Verknüpfung
xor dest, source	dest=dest XOR source	xor eax, ecx Bitweise XOR-Verknüpfung
not op	op=NOT op	not eax Bitweise Invertierung
neg op	op=0-op	neg eax Negativer Wert im Zweierkomplement

4.3 Shift- und Rotationsbefehle

Shift-Befehl

Tabelle 5 zeigt eine ganze Reihe von Shift- und Rotationsbefehlen, die IA-32 beherrscht. Durch diese werden oftmals arithmetische Operationen mit Zweierpotenzen nachgebildet. Shift-Befehle verschieben einen Registerinhalt um eine Anzahl Bits nach links oder rechts, wobei grundsätzlich das zuletzt herausgeschobene Bit im Carryflag gespeichert wird. Beim arithmetischen Rechts-Shift (`sar`) werden keine Nullen nachgeschoben, sondern das MSB (*Most Significant Bit*) wird dupliziert, um das Vorzeichen bei Zweierkomplementdarstellung zu erhalten. Bei den übrigen Shift-Befehlen werden immer Nullen eingeschoben.

Rotationsbefehl

Die Rotationsbefehle rotieren Registerinhalte, d. h. die auf einer „Seite“ des Registers herausgeschobenen Bits werden auf der anderen Seite wieder reingeschoben. Diese Rotation erfolgt entweder durch das Carryflag (`rcl` und `rcr`) oder nicht (`rol` und `ror`). Grundsätzlich steht das zuletzt rotierte Bit jedoch immer im Carryflag. Abb. 6 zeigt noch einmal die Funktionsweisen der unterschiedlichen Shift- und Rotationsbefehle.

Tabelle 5: Shift und Rotationsbefehle

Befehl	Operation	Beispiel
shl op, q	Links-Shift um q Bits LSB = 0 CF = MSB	shl ebx, 4 LSB = least significant bit MSB = most significant bit
shr op, q	Rechts-Shift um q Bits MSB = 0 CF = LSB	shr ebx, 4
sal op, q	Arithmetischer Links-Shift um q Bits identisch mit shl	sal ebx, 4 für Zweierkomplementdarstellung
sar op, q	Arithmetischer Rechts-Shift um q Bits MSB = MSB CF = LSB	sar ebx, 4 für Zweierkomplementdarstellung
rol op, q	Linksrotation um q Bits CF = MSB LSB = MSB	rol ebx, 4
ror op, q	Rechtsrotation um q Bits CF = LSB MSB = LSB	ror ebx, 4
rcl op, q	Linksrotation um q Bits durch CF CF = MSB LSB = CF	rcl ebx, 4
rcr op, q	Rechtsrotation um q Bits durch CF CF = LSB MSB = CF	rcr ebx, 4

K

Kontrollaufgabe 1

1. Welcher Wert steht nach Ausführung dieser beiden Befehle in `eax`?

```
mov eax, 32768
sar eax, 5
```

2. Welcher Wert steht nach Ausführung dieser Befehlsfolge in `eax`?

```
mov eax, 4711h
ror eax, 1
sar eax, 5
rcl eax, 4
shr eax, 4
```

4.4 Kontrolltransferbefehle

Assemblerbefehle werden im Normalfall sequentiell abgearbeitet, also Befehl nach Befehl durch Inkrementierung der Instruction Pointer `eip`. Eine Änderung dieses sequentiellen Kontrollflusses erfolgt durch den Aufruf von Subroutinen, Sprungbefehlen (sogenannte *Jumps*) und Interrupts. Subroutinen werden anschließend in Abschnitt 4.6 und Interrupts in Abschnitt 4.10 behandelt. Wir wenden uns zunächst den Sprungbefehlen zu.

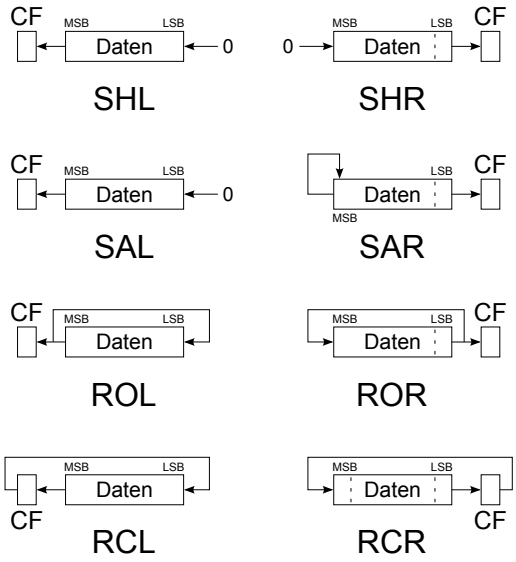


Abb. 6: Shift- und Rotationsbefehle

Es werden grundsätzlich zwei Arten von Sprungbefehlen unterschieden:

a) **Unbedingter Sprung:** Dieser Sprung wird immer ausgeführt. Die Syntax lautet: unbedingter Sprung

```
jmp dest
```

`dest` gibt dabei die Zieladresse (Sprungadresse) im Programmspeicher an. Die Zieladresse muss dabei keine statische Adresse sein. Sie kann auch in einem Register abgelegt sein. Man spricht dann von einer indirekten Adressierung (Beispiel: `jmp [ebx+4]`).

b) **Bedingter Sprung:** Dieser Sprung erfolgt nur, wenn eine Bedingung erfüllt ist. Ansonsten wird mit dem Nachfolgebefehl in der sequentiellen Befehlsreihenfolge fortgefahren. Die Syntax lautet: bedingter Sprung

```
jCC dest
```

Die Bedingung `CC` bezieht sich dabei auf den Status bestimmter Flags im `eflags`-Register, die dazu meist im vorangegangenen Befehl gezielt beeinflusst wurden.

B

Beispiel 4

Die Übersetzung einer *for*-Schleife der Gestalt

```
for (int i=0; i<10; i++) ...
```

könnte in Assembler wie folgt aussehen:

```
mov ecx, 0     ; i=0
loop:
cmp ecx, 10    ; Vergleich i mit 10
jae end        ; jae (jump if above or equal) prueft Flags und
               ; springt zum Ende, falls i >= 10 ist
...            ; hier steht der Code fuer den Schleifenrumpf
inc ecx        ; i++
jmp loop       ; Statischer Sprung zum Beginn der Schleife
end:
```

Viele Befehle beeinflussen die Flags. Vor bedingten Sprüngen wird aber oftmals ein `cmp`-Befehl ausgeführt:

```
cmp op1, op2
```

Der `cmp`-Befehl hat zwei Operanden `op1` und `op2`, aber kein Ziel. Der `cmp`-Befehl entspricht dem `sub`-Befehl (Subtraktion), allerdings mit dem Unterschied, dass hier nur die Flags verändert werden. Das Ergebnis der Subtraktion wird nicht gespeichert, sondern verworfen. (In den folgenden Tabellen wird jeweils die Ausführung eines `cmp`-Befehls vorausgesetzt, obwohl auch andere Befehle die Flags beeinflussen. Aber nur im Fall des `cmp`-Befehls oder des `sub`-Befehls haben die Mnemonics der bedingten Sprungbefehle eine intuitive Bedeutung.)

Für vorzeichenlose und vorzeichenbehaftete Zahlen müssen unterschiedliche Sprungbefehle benutzt werden, die die Flags unterschiedlich berücksichtigen bzw. auswerten.

Sprungbefehle für vorzeichenlose Zahlen

Tabelle 6 zeigt die bedingten Sprünge für vorzeichenlose Zahlen und den Status der Flags, um die jeweiligen Bedingungen zu erfüllen. Wie man in Spalte 3 der Tabelle erkennen kann, ist jeder Vergleich an eine charakteristische Kombination von Carryflag und Zeroflag geknüpft. Wird die vorangegangene Ausführung eines `cmp`-Befehls vorausgesetzt, so zeigt das Zeroflag die Gleichheit der Operanden an. Sind beide Operanden gleich, so ergibt die Subtraktion 0, und das Zeroflag ist gesetzt, andernfalls wird es gelöscht. „Kleiner“ und „größer“ werden durch das Carryflag angezeigt: Wenn Operand1 kleiner ist als Operand2, so wird durch die Subtraktion das Carryflag gesetzt, andernfalls wird es gelöscht.[8]

[8] Zu beachten ist hier, dass das Carryflag so gesetzt wird, als hätte tatsächlich eine binäre Subtraktion stattgefunden. In Wahrheit führt die CPU eine Addition mit dem (im Zweierkomplement) invertierten Wert von Operand 2 durch und invertiert anschließend das Carryflag.

Befehl	Abkürzung für	Zustand der Flags	Bedeutung
ja	jump if above	CF = 0 und ZF = 0	Operand1 >Operand2
jae	j. if above or equal	CF = 0	Operand1 >= Operand2
jb	j. if below	CF = 1	Operand1 < Operand2
jbe	j. if below or equal	CF = 1 oder ZF = 1	Operand1 <= Operand2
je	j. if equal	ZF = 1	Operand1 = Operand2
jne	j. if not equal	ZF = 0	Operand1 != Operand2
jna	j. if not above		identisch jbe
jnae	j. if not above or equal		identisch jb
jnb	j. if not below		identisch jae
jnbe	j. if not below or equal		identisch ja

Tabelle 6: Bedingte Sprünge für vorzeichenlose Zahlen

E

Exkurs 3: **Zweierkomplementdarstellung**

Zur Darstellung vorzeichenbehafteter Zahlen wird üblicherweise die Zweierkomplementdarstellung verwendet. Dadurch ist mit n Bit der Zahlenbereich $-2^{n-1}, ..., 0..., 2^{n-1} - 1$ abbildbar.

Positive Zahlen (inklusive der 0) werden in der Zweierkomplementdarstellung mit einer führenden 0 (Vorzeichenbit) versehen, die übrigen Bits folgen der üblichen Binärdarstellung (z. B. 0100_2 für 4_{10}).

Die Darstellung einer negativen Zahl erfolgt durch bitweise Negation der korrespondierenden positiven Zahl und anschließende Addition einer 1. Das führende Bit ist immer 1. Die Darstellung der -4 unter Verwendung von 8 Bits ist demzufolge:

$$11111100 = 11111011 + 00000001 = not(00000100) + 1$$

(Die Umwandlung von im Zweierkomplement dargestellten negativen Zahlen in positive erfolgt analog.)

Die Addition im Zweierkomplement erfolgt ohne Fallunterscheidung für positive und negative Zahlen. Eine Subtraktion wird auf eine Addition zurückgeführt.

Beispiel für 8-Bit-Zahlen:

$$8_{10} - 12_{10} = 8_{10} + (-12_{10}) = 00001000_2 + 11110100_2 = 11111100_2 = -4_{10}$$

Ein Ergebnis einer Addition, das den Wertebereich überschreitet, erzeugt einen Overflow. Dieser wird durch das Overflowflag angezeigt.

Noch eine allgemeine Bemerkung: Die Interpretation, ob eine n-Bit-Binärzahl mit einer führenden 1 eine negative Zahl in Zweierkomplementdarstellung oder eine positive Zahl $> 2^{n-1} - 1$ ist, obliegt dem Programmierer. Der Programmierer ist damit auch für die richtige Interpretation und die richtige Auswertung der Flags im jeweiligen Kontext verantwortlich.

Sprungbefehle für vorzeichenbehaftete Zahlen

Die bedingten Sprünge nach einem `cmp`-Befehl mit vorzeichenbehafteten Zahlen sind in Tabelle 7 aufgeführt. Die Befehle `je` und `jne` sind bei vorzeichenlosen und vorzeichenbehafteten Operanden identisch und tauchen deshalb in der Tabelle nicht noch einmal auf.

Tabelle 7: Bedingte Sprünge für vorzeichenbehaftete Zahlen

Befehl	Abkürzung für	Zustand der Flags	Bedeutung
jg	j. if greater	ZF = 0 und SF = OF	Operand1 > Operand2
jge	j. if greater or equal	SF = OF	Operand1 >= Operand2
jl	j. if less	SF <> OF	Operand1 < Operand2
jle	j if less or equal	ZF = 1 oder SF <> OF	Operand1 <= Operand2
jng	j if not greater		identisch jle
jnge	j if not greater or equal		identisch jl
jnl	j if not less		identisch jge
jnle	j if not less or equal		identisch jg

K

Kontrollaufgabe 2

Erklären Sie anhand von vier Beispielen den Status der Flags bei bedingten Sprüngen nach einem `cmp`-Befehl mit vorzeichenbehafteten Operanden in Zweierkomplementdarstellung. Benutzen Sie dazu der Einfachheit halber 8-Bit-Zahlen. Die Beispiele sollten alle vier möglichen Kombinationen von positiven und negativen Operanden beim `cmp`-Befehl beinhalten.

Neben den bisher aufgelisteten bedingten Sprungbefehlen existiert noch eine ganze Reihe weiterer, die sich explizit auf den Status einzelner Flags beziehen. Diese sind in Tabelle 8 aufgelistet.

Tabelle 8: Bedingte Sprünge bezogen auf einzelne Flags

Befehl	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
jnp	jump if not parity	PF = 0
jpe	jump if parity even	PF = 1
jpo	jump if parity odd	PF = 0
jz	jump if zero	ZF = 1
jnz	jump if not zero	ZF = 0
js	jump if sign (<0)	SF = 1
jns	jump if no sign (>=0)	SF = 0

E

Exkurs 4: **Befehlssynonyme**

Es fällt auf, dass sowohl die Befehle `jb` und `jae` als auch die Befehle `jc` und `jnc` testen, ob das Carryflag 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 die Befehle `jp` und `jpe`, auch diese sind Synonyme.

Eine ähnliche Funktion wie der `cmp`-Befehl hat der `test`-Befehl:

```
test op1, op2
```

Auch dieser schreibt kein Ergebnis zurück, sondern führt nur eine AND-Verknüpfung der Operanden durch und setzt die Flags entsprechend.

B

Beispiel 5

Mit dem `test`-Befehl kann bspw. ein bedingter Sprung davon abhängig gemacht werden, ob ein bestimmtes Bit eines Registers gesetzt ist oder nicht.

Bei der Befehlsfolge

```
test eax, 0008h
jnz  ziel
```

wird der Sprung ausgeführt, wenn Bit 3 von `eax` gesetzt ist. Ist Bit 3 von `eax` gesetzt, dann ist das Ergebnis des Tests `0008h`, ansonsten `0000h`. Man spricht auch von einer Maskierung von `eax` durch den Wert `0008h`.

4.5 Speicher und Adressierung

Bisher haben wir die Datentransferbefehle, arithmetischen und logischen Befehle und Kontrolltransferbefehle von IA-32 kennengelernt. Bis auf die Befehle zum Aufruf von Subroutinen und der damit verbundenen Stack-Verwaltung sind das die wesentlichen Kernbefehle dieser Architektur. Zunächst müssen wir uns nun näher mit dem Hauptspeicher und den Adressierungsarten des Hauptspeichers beschäftigen.

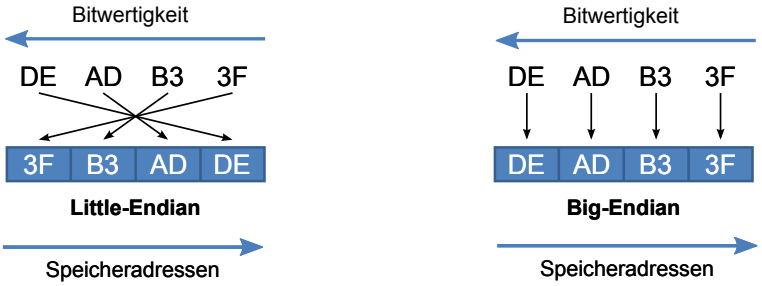
Es reicht im Moment, sich den Speicher als einen linearen, zusammenhängenden Adressraum vorzustellen. Die Startadresse ist die 0, die höchstmögliche Adresse ist $2^{32} - 1$, da ein 32 Bit breiter Adressbus vorliegt. Adressiert werden einzelne Bytes, wodurch sich eine Maximalgröß des Speichers von 4 GB (GB = Gigabyte) ergibt. Innerhalb des Speichers gibt es verschiedene Bereiche, in denen entweder Code oder Daten liegen. Manche Speicherbereiche können auch unbelegt sein. Man sagt, auf diese Bereiche ist nichts *gemappt*.

Endianness

Zunächst ist es wichtig zu wissen, *wie* Daten im Hauptspeicher abgelegt werden. Die Anordnung der Bytes eines z. B. 32-Bit-Datums (4 Byte) im Hauptspeicher bezeichnet man als *Endianness*. Es gibt zwei Möglichkeiten: Entweder wird das niederwertigste Byte an der kleinsten (der vier hintereinander liegenden) Speicheradressen abgelegt und das höchstwertige Byte an der höchsten Adresse oder umgekehrt. Im ersten Fall spricht man von *Little Endian*, im zweiten Fall von *Big Endian*. Abb. 7 zeigt dies beispielhaft an einem Datum `DEADB33Fh`:

Die x86-Prozessorfamilie verwendet Little Endian. Für den menschlichen Betrachter ist das etwas gewöhnungsbedürftig, weil die Bytes scheinbar verdreht im Speicher liegen, wenn man sich einen Speicherbereich mit einem Hex-Editor anschaut. Dem Prozessor ist es egal, er muss sich beim Schreiben und Lesen nur konsequent an diese Konvention halten.

Abb. 7: Endianness



IA-32 bietet vier verschiedene Möglichkeiten der Adressierung:

1. **Immediate**
2. **Register**
3. **Memory**
4. **Offset**

Wie schon erwähnt darf pro Befehl in der Regel nur eine Hauptspeicherreferenzierung stattfinden. Es ist also nicht möglich, ein Datum direkt von einer Speicheradresse in eine andere zu kopieren. Es ist dazu notwendig, ein Register als Zwischenspeicher zu benutzen.

B

Beispiel 6

Kopieren zwischen Adressen im Hauptspeicher:

```
mov [123], [456]     ; funktioniert nicht,
                     ; Umweg ueber Register noetig.

mov eax, [456]
mov [123], eax       ; funktioniert!
```

Tabelle 9 zeigt die vier genannten Adressierungsarten anhand von Beispielen.

Tabelle 9: Adressierungsarten von IA-32

Befehl	Adressierung	Bedeutung
mov eax, 1337	Immediate	Laden von Konstante 1337 nach eax
mov eax, ebx	Register	Laden von ebx nach eax
mov eax, [1234]	Memory	Laden von Speicherinhalt (statische Speicheradresse)
mov esi, [edi]	Offset	Laden von Speicherinhalt mittels Zeiger (*Pointer*) in Register

mov eax, [ebx+ecx]
mov eax, [ebx+1234]
mov eax, [ebx+ecx*4-1337]
	Offset	Erweiterter Zugriff mit Index Displacement und Scale

Der Hauptspeicherzugriff wird immer durch eckige Klammern dargestellt. Daraus kann abgeleitet werden, dass es sich bei den Adressierungsarten *Immediate* und *Register* nicht um Adressierungsarten des Hauptspeichers handelt. Sie werden aber trotzdem zu den Adressierungsarten gezählt.

Immediate-Adressierung

Die Immediate-Adressierung dient zum Laden von Konstanten. Die Konstante steht unmittelbar im Befehl. Er wird bei der Übersetzung fest in den Maschinencode eingebaut. So wird bspw. aus dem Assemblerbefehl `mov al,9` der Maschinencode `B009`.

Bei der Registeradressierung sind Quelle und Ziel interne Register des Prozessors. Registeradressierung

Bei der Memory-Adressierung bzw. direkten Speicheradressierung wird ein Platz im Hauptspeicher direkt angesprochen. Diese Adresse muss zur Übersetzungszeit feststehen. Sie wird in den Maschinencode fest eingebunden. Memory-Adressierung

B

Beispiel 7

Addition zweier Variablen: a = a + b

```
mov eax, [1337] ; Variable a liegt in Speicherzelle 1337
mov ecx, [1234] ; Variable b liegt in Speicherzelle 1234
add eax, ecx
mov [1337], eax ; Ergebnis wird zurueck nach a kopiert

; oder alternativ

mov eax, 1337 ; Adresse von a wird in eax kopiert
mov ecx, [1234]
add [eax], ecx ; Ergebnis wird direkt in a gespeichert
```

Wie in der 2. Variante zu sehen ist, sind Speicherzugriffe nicht nur mit dem `mov`-Befehl möglich, sondern auch mit anderen Befehlen, die mit Operanden arbeiten. IA-32 ist keine Load-/Store-Architektur, d. h. es gibt keine dedizierten Schreib-/Lesebefehle für den Speicher. Des Weiteren fällt in diesem Beispiel auf, dass durch `add [eax], ecx` zwei Speicherzugriffe stattfinden, zuerst ein lesender und dann ein schreibender. Da jeweils dieselbe Adresse im Speicher referenziert wird, ist das in diesem Fall erlaubt.

Assembler erlauben die Verwendung von Variablennamen für feste Speicheradressen. Wenn im vorangegangenen Beispiel ein Assembler den Variablennamen `a` und `b` die Adressen `1337` bzw. `1234` zugeordnet hat, so kann die zweite Variante des Programmfragments wie folgt aussehen, wobei die Verwendung der eckigen Klammern um `a` und `b` optional ist:

```
mov eax, a
mov ecx, [b]
add [eax], ecx
```

Eine Variable kann zudem mit einem konstanten Offset versehen sein (z. B. `mov eax,[basis+4]`), um bspw. feste Feldelemente zu adressieren.

Bei der Offset-Adressierung können Speicheradressen dynamisch aus mehreren Registern berechnet werden. Das Ergebnis der Berechnung ist die *effektive Adresse* der Speicherzelle, die angesprochen werden soll. IA-32 bietet mehrere Möglichkeiten der Offset-Adressierung mit unterschiedlicher Komplexität an: Offset-Adressierung

1. **Base**:
   Hier wird der Inhalt eines Registers als Pointer auf eine Adresse verwendet.

B

Beispiel 8

```
mov edi, 123    ; Speichert 123 in edi
mov [edi], 1337 ; Kopiert Konstante 1337 an die in edi
                ; gespeicherte Adresse, also 123.
```

äquivalent dazu ist:

```
mov [123], 1337
```

2. **Base + Index**:
   Der Inhalt zweier Register (Base und Index) wird aufaddiert und das Ergebnis als Pointer benutzt.

B

Beispiel 9

```
mov ebx, 123
mov ecx, 7
mov [ebx+ecx], 1337 ; Speichert 1337 unter Adresse 130
```

Als Indexregister können alle General Purpose Register verwendet werden.

3. **Base + Index * Scale** (Array-Zugriff):
   Die Scale-Konstante wird für Array-Zugriffe benutzt. Der Wert kann 1, 2, 4 oder 8 betragen und entspricht der Größe des verwendeten Datentyps in Byte. Für ein Integer Array wird z. B. 4 verwendet.

B

Beispiel 10

Code in C:

```
int a[256]; return a[i];
```

Code in Assembler:

```
  ; base-a sei die Base-Adresse von Array a
  ; Variable i enthalte den Index
mov ebx, base-a       ; Base-Adresse von a wird
                      ; nach ebx kopiert
mov ecx, [i]          ; Wert von i wird
                      ; nach ecx kopiert
mov eax, [ebx+ecx*4]  ; i-tes Element von a wird
                      ; nach eax kopiert
```

4. **Displacement-Operator**:
   Auf die aus Base, Index und Scale berechnete Adresse lässt sich noch ein konstanter Wert, ein sogenannter Displacement-Wert, aufaddieren:

B

Beispiel 11

```
mov eax, [eax+ecx*4+4711h]
```

Diese Zugriffsart ist sehr kompakt und typisch für CISC-Architekturen. In vielen anderen Architekturen müsste die effektive Adresse in mehreren

Schritten durch Additions- und Multiplikationsbefehle berechnet werden. Abb. 8 zeigt eine Übersicht der Offset-Adressierungsmöglichkeiten mit den für Base und Index zugelassenen Registern.

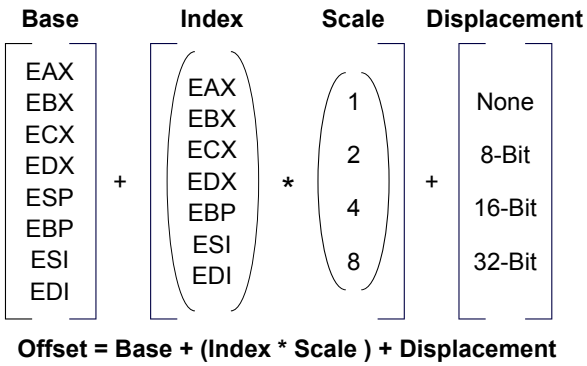


Abb. 8: Offset-Adressierung im Überblick

Der Vollständigkeit halber muss an dieser Stelle noch der Befehl `lea` (*load effective Address*) genannt werden.

Der Befehl

```
lea dest, source
```

berechnet die effektive Adresse von `source` und speichert diese in `dest`. `lea` gibt also eine Speicheradresse zurück, auf die dann andere Befehle zugreifen können.

Beispiel 12

Die Codesequenz

```
lea ecx, [eax+ecx*4+4711h]
mov eax, [ecx]
```

ist äquivalent zu:

```
mov eax, [eax+ecx*4+4711h]
```

B

4.6 Subroutinen

In diesem Abschnitt wird erläutert, wie Funktionen und Funktionsaufrufe in Assembler realisiert werden. Der Begriff Funktion ist hierbei gemäß der C-Konvention im Sinne von Unterprogramm oder Subroutine zu verstehen. Die grundlegenden Befehle sind in Tabelle 10 zusammengefasst und werden im Verlauf dieses Abschnitts näher erklärt.

Für die Implementierung von Subroutinen ist die Funktionsweise des Stack von grundlegender Bedeutung. Diese wird im Folgenden detailliert beschrieben.

Stack

Der Stack ist die zentrale Datenstruktur, die beim Aufruf von Subroutinen verwendet wird. Über den Stack werden Parameter an die aufgerufene Funktion

Stack

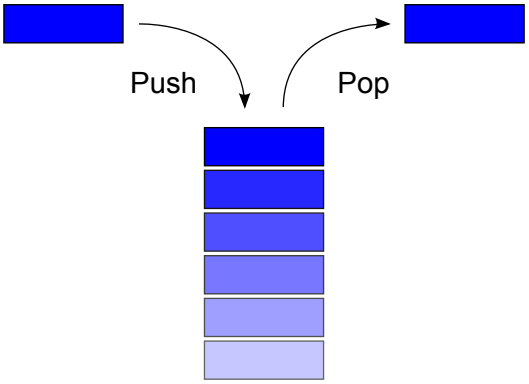
Tabelle 10: Subroutinen-Befehle

Befehl	Operation	Beispiel
call sub	dec esp [esp]=eip eip=sub	call 1234 Rueckspungadresse auf Stack Sprung in Subroutine
ret (op)	eip=[esp] inc esp (add esp,op)	ret 8 Rueckspungadresse vom Stack Stack um 8 Byte kuerzen
push op	dec esp [esp]=op	push eax eax auf Stack ablegen
pop op	op=[esp] inc esp	pop eax eax von Stack laden
pushf	push flags	
popf	pop flags	
pushad	push all general regs	
popad	pop all general regs	

übergeben, die lokalen Variablen der Funktion sowie die Aufrufinformationen werden dort abgelegt. Dafür steht ein dedizierter Speicherbereich bereit, und die Verwaltung des Stack wird von der Hardware direkt unterstützt. IA-32 spezifiziert hierfür eigene Register (`esp` und `ebp`) und Befehle.

Der Stack wird über die beiden Befehle `push` und `pop` kontrolliert. `push` legt einen 16- oder 32-Bit-Wert auf dem Stack ab, `pop` nimmt ihn vom Stack (Abb. 4.6).

Abb. 9: Stack-Prinzip mit push und pop



Der Stack „wächst“ bei IA-32 von oben nach unten, also von höheren Speicheradressen hin zu niedrigeren. Der Stack Pointer `esp` zeigt immer auf das „oberste“ Element des Stack, genauer gesagt auf das oberste Byte. Diese Speicherstelle wird auch als TOS (*Top of Stack*) bezeichnet.

Bemerkung: Wir verwenden hier immer den Ausdruck „oberstes“ Element, obwohl der Stack im Speicher von oben nach unten – also von größeren Speicheradressen zu kleineren – wächst. Gemeint ist das letzte Byte auf dem Stack.

Durch den `push`-Befehl wird ein Datum, also z. B. ein Registerinhalt auf den Stack abgelegt. Gespeichert wird der Wert unmittelbar oberhalb von TOS. `esp` wird um 2 oder 4 dekrementiert (für 16- bzw. 32-Bit-Werte) und zeigt somit auf den neuen

TOS. Beim `pop`-Befehl läuft der Vorgang umgekehrt ab: Der Wert wird vom Stack gelesen und `esp` wird inkrementiert.

Kontrollaufgabe 3

K

Skizzieren Sie den Byte-weisen Aufbau des Stack nach dem zweiten `push`-Befehl. Beachten Sie dabei Little Endianness.

```
mov  eax, 4711h
mov  ebx, 0815h
push ebx
push eax
pop  eax
pop  ebx
```

Kontrollaufgabe 4

K

Was fällt bei dieser Befehlsfolge auf?

```
mov  eax, 4711h
mov  ebx, 0815h
push ebx
push eax
pop  esp
pop  ebx
```

Hinweis: Nicht alle Register sind als Operanden bei `push` und `pop` erlaubt.

Stack-Verwaltung

Die „Ordnung“ bei der Stack-Verwaltung obliegt weitgehend dem Programmierer. `push`- und `pop`-Befehle sollten ausbalanciert sein, d. h. zu jedem `push` gehört ein `pop` in umgekehrter Reihenfolge. Insbesondere sollte nach Beendigung eines Unterprogrammaufrufs der Stack wieder so aussehen wie vor dem Aufruf.

Unabhängig von Unterprogrammaufrufen kann der Stack zur Ablage von Zwischenergebnissen bei irgendwelchen Berechnungen benutzt werden. Dies wird auch sehr häufig gemacht und hat den Vorteil, dass der Programmierer sich nicht um die genauen Speicheradressen kümmern muss, sondern nur um die allgemeine Stack-Ordnung.

Unterprogramme

Der Aufruf eines Unterprogramms verändert den Kontrollfluss eines Programms. Das aufrufende Programm wird unterbrochen, das Unterprogramm wird ausgeführt, und das unterbrochene Programm wird anschließend fortgesetzt. Die zentralen Befehle bei einer Unterprogrammausführung sind `call` und `ret`. Sie bedienen sich des Stack.

Sei `f` die Adresse eines Unterprogramms, dann ist der Befehl

```
call f
```

in etwa logisch äquivalent zu dieser Befehlssequenz:

```
push eip
jmp  f
```

Der aktuelle Instruction Pointer `eip` wird auf den Stack gelegt[9] und anschließend erfolgt ein unbedingter Sprung nach `f`. Der Rücksprung aus dem Unterprogramm erfolgt mit dem Befehl `ret` (für *return*), der den ursprünglichen `eip` wiederherstellt (`pop eip`).

Das Ablegen und Rückladen des `eip` ist notwendig, weil Unterprogramme von unterschiedlichen Stellen des übergeordneten Programms aus aufgerufen werden könnten. Die Rücksprungadresse ist daher dynamisch und nicht statisch vorhersagbar.

Parameter

An Unterprogramme werden in der Regel Parameter übergeben, und das Unterprogramm liefert Ergebnisse zurück. Auch diese Parameter- und Ergebnisübergaben erfolgen dynamisch über den Stack. Globale Parameteradressen sind nur in Ausnahmefällen sinnvoll. Die Übergabe von Parametern an eine Subroutine veranschaulicht das folgende Beispiel:

B

Beispiel 13

Funktionsaufruf: F(1,2,3)

Implementierung in Assembler:

```
push 3
push 2
push 1
call F
```

Es werden nacheinander die Parameter 3, 2 und 1 auf dem Stack abgelegt, danach der `eip` mittels `call`.

Stack Frame

Die Adressierung innerhalb des Stack erfolgt über den Stack Pointer `esp` und den Base Pointer `ebp`. `ebp` zeigt (*vereinbarungsgemäß*, weil seine Verwaltung allein in den Händen des Programmierers liegt) auf den Anfang eines *Stack Frame*. Ein Stack Frame ist ein Teil des gesamten Stack, der einem Teilprogramm (Hauptprogramm oder Unterprogramm) zugeteilt ist.

Beim Aufruf eines Unterprogramms wird ein neuer Stack Frame oben auf dem aktuellen Stack angelegt. Dazu wird der aktuelle `esp` nach `ebp` kopiert. `ebp` zeigt somit auf den Anfang des neuen Stack Frame. Zuvor muss aber der alte Base Pointer `ebp` gerettet werden, sonst ginge er für das aufrufende Programm verloren. Er wird mittels `push` auf den Stack gelegt. `ebp` wird innerhalb des Unterprogramms nicht verändert. Über ihn werden mittels Offsets Parameter und lokale Variablen adressiert.

Vor Beendigung des Unterprogramms wird `ebp` nach `esp` zurück kopiert, der alte `esp` wird also wiederhergestellt. Dann wird noch der alte `ebp` mittels `pop` regeneriert. Damit ist der Stack Frame des Unterprogramms vom Stack verschwunden.

[9] Es wird in Wahrheit die Adresse des Befehls nach dem `call`-Befehl auf den Stack gelegt, denn an dieser Stelle soll ja das unterbrochene Programm später fortgesetzt werden.

Abbildung 10 zeigt den Auf- und Abbau eines Stack Frame mit den Veränderungen der Stack-Register `esp` und `ebp`.

Prolog und Epilog

Diesen allgemeinen Ablauf zu Beginn und am Ende eines Unterprogramms nennt man *Prolog* bzw. *Epilog*. Während des Prologs werden auch Bereiche des Stack Frame für lokale Variablen reserviert. Das folgende Beispiel zeigt einen typischen Aufbau von Prolog und Epilog in Assembler. Neben der bereits ausführlich beschriebenen Behandlung von Stack Pointer und Frame Pointer werden hier 16 Byte im Stack Frame für lokale Variablen reserviert, indem `10h` vom `esp` subtrahiert wird.

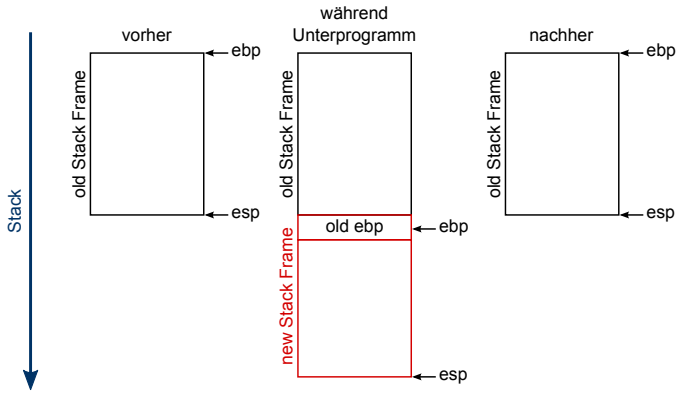


Abb. 10: Auf- und Abbau eines Stack Frame

B

Beispiel 14

Typischer Prolog/Epilog einer Subroutine

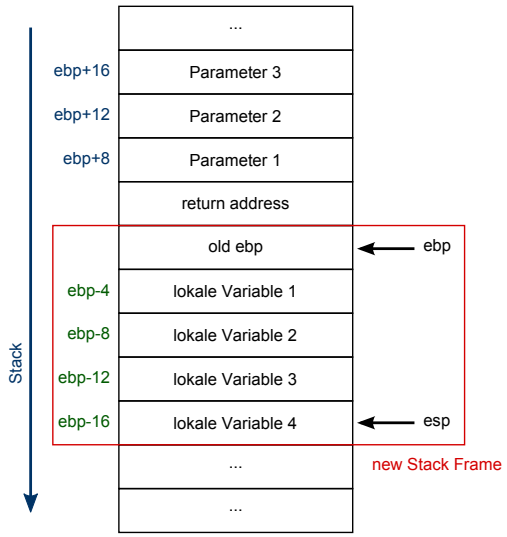
```
push ebp       ; ebp wird auf den Stack gesichert.
mov ebp, esp   ; Alter Wert von esp wird in ebp gesichert.
sub esp, 10h   ; Subtraktion von 16 vom Stack Pointer: lokaler
               ; Variablenbereich von 16 Byte wird reserviert.
.
.              ; eigentlicher Code der Subroutine
.
mov esp, ebp   ; Wiederherstellung des alten Wertes von esp:
               ; dadurch werden die lokalen Variablen aus dem
               ; Stack-Bereich genommen.
pop ebp        ; Wiederherstellung von ebp.
ret            ; Ruecksprung zur aufrufenden Funktion.
```

Wenn wir davon ausgehen, dass die aufrufende Funktion unmittelbar vor dem Funktionsaufruf Übergabeparameter auf den Stack gelegt hat, dann sieht die Speichersituation auf dem Stack nach Ablauf des Prologs im Beispielprogramm wie in Abb. 11 dargestellt aus. Im Stack liegen hintereinander die Parameter und der gesicherte `eip` der aufrufenden Funktion. Es folgt der Stack Frame der aufgerufenen Funktion mit dem gesicherten `ebp` und den lokalen Variablen.

Referenzierung von lokalen Variablen und Parametern

Lokale Variablen und Parameter können innerhalb der Funktion wie folgt referenziert werden:

Abb. 11: Aufbau des Stack Frame nach dem Prolog



- `[ebp-x]` referenziert lokale Variablen wobei x >= 4 ist
- `[ebp+x]` referenziert Parameter wobei x >= 8 ist

Das folgende Beispiel zeigt die Übersetzung einer kompletten C-Funktion mit Parametern und lokalen Variablen nach diesem Schema.

B

Beispiel 15

Code in C:

```
void F(int param1, int param2) {
  int var1 = 0, var2 = 1;
  param1 = var1;
  param2 = var2;
}
```

Code in Assembler:

```
push ebp
mov ebp, esp
sub esp, 8            ; Reservierung von 8 Byte fuer 2 Integer
mov [ebp-4], 0        ; [ebp-4] referenziert var1
mov [ebp-8], 1        ; [ebp-8] referenziert var2
mov eax, [ebp-4]      ;
mov [ebp+8], eax      ; [ebp+8] referenziert param1
mov ecx, [ebp-8]      ;
mov [ebp+12], ecx     ; [ebp+12] referenziert param2
mov esp, ebp
pop ebp
ret
```

Kontrollaufgabe 5

Zeichnen Sie detailliert den Aufbau und den Inhalt des Stack beim Ablauf der Beispielfunktion unmittelbar vor dem Befehl `mov esp,ebp`.

K

Register, die in einer Subroutine verändert werden, müssen nach Ablauf der Subroutine in der Regel wieder in den Ausgangszustand zurückgesetzt werden. Dies kann entweder im aufrufenden Programm oder im Prolog/Epilog des Unterprogramms erfolgen.

Beispiel 16

Hier übernimmt das Unterprogramm die Rettung der Register. Dazu werden sie im Prolog auf dem Stack gespeichert und im Epilog in umgekehrter Reihenfolge wieder vom Stack genommen.

```
push ebx           ; Prolog: Sicherung der verwendeten
push esi           ; Register
push ebp
mov ebp, esp
.
mov ebx, 0         ; Verwendung ebx
.
mov esi, DEADBEEFh ; Verwendung esi
.
pop ebp            ;
pop esi            ; Epilog: Wiederherstellung der
pop ebx            ; Register (umgekehrte Reihenfolge)
ret
```

B

Kontrollaufgabe 6

Nehmen Sie noch einmal das Unterprogramm:

```
void F(int param1, int param2)
{
int var1 = 0, var2 = 1;
param1 = var1;
param2 = var2;
}
```

Übersetzen Sie es wie im Beispiel davor gezeigt in Assembler. Es sollen dabei alle verwendeten Register im Prolog gesichert und im Epilog wiederhergestellt werden. Es ist insbesondere auf die korrekte Referenzierung der Parameter und Variablen zu achten.

K

Aufrufkonventionen

Aufrufkonventionen definieren, wie Parameter an Subroutinen übergeben werden und inwiefern der Registerzustand nach Beendigung der Subroutine wiederhergestellt werden muss. Innerhalb seines eigenen Programms obliegt es dem Programmierer selbst, wie dies geschieht. Beim Aufruf von externem Code – insbesondere bei vorgefertigten Bibliotheksfunktionen – müssen sich aufrufende und aufgerufene Funktion aber an dieselbe Konvention halten.

Aufrufkonventionen geben Antwort auf folgende Fragen:

- Wo werden Parameter abgelegt? (Stack oder Register)
- Wer bereinigt den Stack von Parametern? (die aufrufende oder die aufgerufene Funktion?)
- Welche Register müssen nach dem Unterprogrammlauf wiederhergestellt werden?
- In welcher Reihenfolge werden Parameter auf dem Stack abgelegt?

Bibliotheksfunktionen

Bibliotheksfunktionen sind oftmals Betriebssystemfunktionen oder sie müssen zumindest vom Betriebssystem bei der Prozesserstellung in Programme eingebunden werden. Aus diesem Grund gibt das Betriebssystem letztendlich auch die Konvention von Unterprogrammaufrufen vor. Unter dem Betriebssystem Windows sind die folgenden Aufrufkonventionen üblich:

- **cdecl**: Die Standardkonvention von C
  Alle Parameter werden über den Stack übergeben, und der aufrufende Code bereinigt nach Ausführung der Subroutine den Stack. Dadurch ist eine variable Parameteranzahl möglich (z. B. bei der C-Funktion `printf`).
- **stdcall**: Die Standardkonvention der Windows API
  Der Unterschied zu *cdecl* besteht darin, dass die aufgerufene Funktion den Stack bereinigt. Dies bedeutet, dass zur Kompilierungszeit der aufgerufenen Funktion die Anzahl der Parameter bekannt sein muss. Infolgedessen sind mit *stdcall* keine Funktionen mit variabler Parameteranzahl möglich.
- **fastcall**: Häufig benutzte Konvention im Betriebssystem und bei Treibern
  Die Parameterübergabe erfolgt teilweise über Register. Diese Konvention heißt „schnell“ (*fast*), da weniger Speicherzugriffe als bei einer Parameterübergabe über den Stack nötig sind.
- **thiscall**: Konvention für Klassenmethoden in C++
  Der Methodenaufruf beinhaltet immer einen Verweis auf das betroffene Objekt. Dieser *this-Pointer* wird in `ecx` übergeben, ansonsten ist *thiscall* identisch mit *stdcall*. (Falls eine variable Parameterzahl notwendig ist, wird die Konvention *cdecl* verwendet, und der this-Pointer wird als letztes auf den Stack gelegt.)

Tabelle 11 zeigt zusammengefasst die wesentlichen Merkmale der vier genannten Aufrufkonventionen. Reservierte Register sind solche, die möglicherweise innerhalb der Funktion verändert werden. Das aufrufende Programm muss sie daher ggf. selbst sichern.

Tabelle 11: Aufrufkonventionen

	cdecl	stdcall	fastcall
Parameter - übergabe	Stack	Stack	Erste 3 Argumente in eax, edx, ecx, Rest per Stack
Zurücksetzen der Parameter	Aufrufender Code	Aufgerufene Funktion	Aufgerufene Funktion
Reservierte Register	eax, ecx, edx	eax, ecx, edx	eax, ecx, edx
Rückgabewert in	eax	eax	eax
Vorteil	Variable Parameterzahl	Geringere Codegröße	schnell

Abb.12 zeigt die Implementierung dieser Aufrufkonventionen in Assemblercode für einen Funktionsaufruf `F(1,2,3)`. Allen Aufrufkonventionen ist gemein, dass Parameter in umgekehrter Reihenfolge auf den Stack gelegt werden, also der rechte Parameter zuerst. Bei *cdecl* bereinigt der aufrufende Code den Stack, löscht also die Parameter vom Stack. Dies geschieht durch den Befehl `add esp,12` nach dem Funktionsaufruf. Wenn man davon ausgeht, dass jeder Parameter vier Byte auf dem Stack belegt, so zeigt der Stack Pointer nach dem `add`-Befehl wieder auf die Position oberhalb der Parameter.

Bei *stdcall* muss die aufgerufene Funktion die Parameter vom Stack nehmen. Dies ist aber problematisch, weil noch die benötigte Rücksprungadresse unterhalb der Parameter auf dem Stack liegt. Bei IA-32 gibt es dazu den speziellen Befehl `ret x` (im Beispiel `ret 12`), der gleichzeitig die Rücksprungadresse vom Stack liest und den Stack Pointer um den Wert `x` erhöht. Im Beispiel wird also 12 auf `esp` addiert und somit der Stack von den Parametern bereinigt.

cdecl	stdcall	fastcall

Aufruf:
`push 3`
`push 2`
`push 1`
`call F`
`add esp, 12`
`cmp eax, 0`
`jz error`
	Aufruf:
	`push 3`
	`push 2`
	`push 1`
	`call F`
	`cmp eax, 0`
	`jz error`
		Aufruf:
		`mov ecx, 3`
		`mov edx, 2`
		`mov eax, 1`
		`call F`
		`cmp eax, 0`
		`jz error`

Subroutine F:
`push ebp`
`mov ebp, esp`
…
`pop ebp`
`ret`
	Subroutine F:
	`push ebp`
	`mov ebp, esp`
	…
	`pop ebp`
	`ret 12`
		Subroutine F:
		`push ebp`
		`mov ebp, esp`
		…
		`pop ebp`
		`ret`

Abb. 12: Implementierung der Aufrufkonventionen in Assembler

4.7 Speicherverwaltung

Privilegierung und Ringe

Mit der 80286-CPU wurde ein mehrstufiges Privilegiensystem eingeführt. Diese Privilegienstufen werden auch als *Ringe* bezeichnet. Das Ausführen von Befehlen und der Zugriff auf Speicherbereiche ist an dieses Privilegiensystem gekoppelt. Beispielsweise sind bestimmte Befehle nur in bestimmten Ringen ausführbar. Ebenso kann der Zugriff auf einzelne Speicherbereiche nur für bestimmte Ringe erlaubt werden.

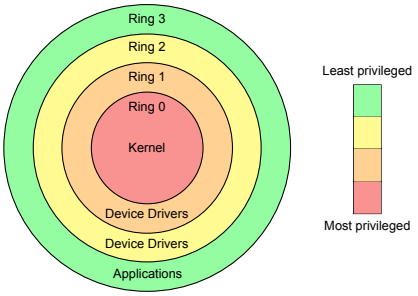
Der Zweck dieses Ringsystems ist es, Stabilität und Sicherheit des Systems zu verbessern. Beispielsweise möchte man verhindern, dass ein beliebiges Anwendungsprogramm in den Speicherbereich der Netzwerkkarte schreiben kann, oder dass bei einem Programmfehler das Betriebssystem komplett abstürzt.

Wie in Abbildung 13[10] dargestellt, besteht das Privilegiensystem aus vier Ringen. Die meisten Betriebssysteme, unter anderem Windows und Linux, benutzen davon allerdings nur die Ringe 0 und 3. Im privilegiertesten Ring 0 werden der *Kernel* (Kern des Betriebssystems) und die *Treiber* (Software zur Steuerung von Hardware-Komponenten) ausgeführt. Ring 3 wird für Anwendungen benutzt. Der Übergang von Ring 3 in Ring 0 ist dabei nur über Syscalls, Interrupts und Exceptions möglich.

[10] Quelle: Wikipedia

Der aktuelle Ring ist stets in den unteren beiden Bits des Codesegment-Registers `cs` gespeichert. Dieses Privilegiensystem ist eng mit der Speicherverwaltung verzahnt, die im Folgenden detailliert beschrieben wird.

Abb. 13: Privilegienringe



Adressräume

Um die Speicherverwaltung von IA-32 zu verstehen, müssen zunächst die Sicht auf den Hauptspeicher von unterschiedlichen Betrachtungsebenen und die daraus folgenden Begrifflichkeiten geklärt werden.

Physikalischer Adressraum

Der Hauptspeicher wird immer über eine Adresse angesprochen und liegt damit in einem bestimmten Adressraum. Auf Hardware-Ebene spricht man vom *physikalischen Adressraum*. Der Hauptspeicher besteht aus Speicherbausteinen und wird über den Adressbus angesprochen. Mit jeder Adresse ist eine Zelle dieses Speichers fest verbunden.

Logischer Adressraum

Den Programmierer interessieren der Aufbau und die Adresszuordnung auf Hardware-Ebene primär nicht. Programme arbeiten im sogenannten *logischen Adressraum*. Logische Adressen sind „intuitive“ Adressen bzw. effektive Adressen, um im Assemblervokabular zu bleiben. Alle in den bisher gezeigten Programmsequenzen verwendeten Adressen waren logische Adressen, wie z. B. `[1234]`. Wo genau die Speicherzelle mit der logischen Adresse `1234` im Hauptspeicher liegt, interessiert nicht weiter.

Programme bestehen aus dem eigentlichen Programmcode und irgendwelchen Daten. Bei der Programmausführung werden zudem dynamisch Daten erzeugt und im Speicher abgelegt. Bisher haben wir – ohne dies explizit zu erwähnen oder zu bemerken – drei logische Speicherbereiche *nebeneinander* verwendet: Den Codebereich, den Datenbereich und den Stack-Bereich. Wir arbeiteten also mit drei unterschiedlichen Speichersegmenten und innerhalb dieser Segmente möglicherweise sogar mit denselben logischen Adressen.

Linearer Adressraum

Die bisher logisch parallel existierenden Segmente müssen linearisiert, also auf einen linearen Speicher mit durchgehender Adressierung abgebildet werden. Dazu verwendeten die x86-Prozessoren schon immer die bereits erwähnten Segmentregister. Stark vereinfacht ausgedrückt gibt ein Segmentregister eine Adresse im linearen oder virtuellen Adressraum an, ab der das entsprechende Segment zu finden ist. Die Adresse im linearen Adressraum errechnet sich dann aus der Adresse im Segmentregister und der logischen Adresse (Abb. 14).

Der lineare Adressraum muss nun keineswegs mit dem physikalischen Adressraum übereinstimmen, auch wenn das bei den ersten x86-Prozessoren wie dem 8086-Prozessor tatsächlich der Fall war. Der lineare Adressraum kann bspw. viel größer sein als der physikalische. Der 80386-Prozessor kann 4 GB Hauptspeicher

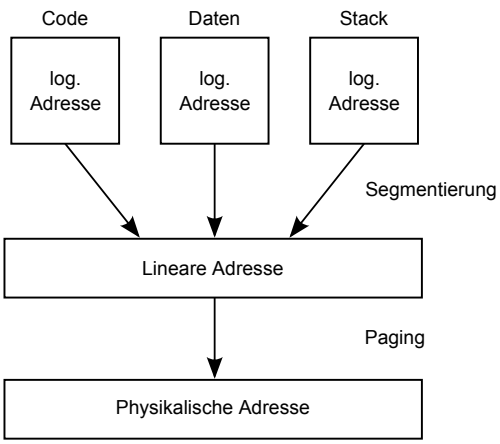


Abb. 14: Logischer, linearer und physikalischer Adressraum

adressieren, zur Zeit seines Erscheinens war man aber noch weit davon entfernt, 4 GB RAM in einen PC einzubauen. Die Speicherverwaltung MMU (*Memory Management Unit*) von IA-32 übernimmt die Abbildung des linearen auf den physikalischen Adressraum, das sogenannte *Mapping*. Hierbei werden Blöcke oder Seiten (*Pages*) fester Größe des linearen Adressraums auf gleichgroße physikalische Speicherseiten abgebildet. Dieses *Paging* erfolgt transparent, d. h. der Programmierer muss sich darum nicht kümmern.

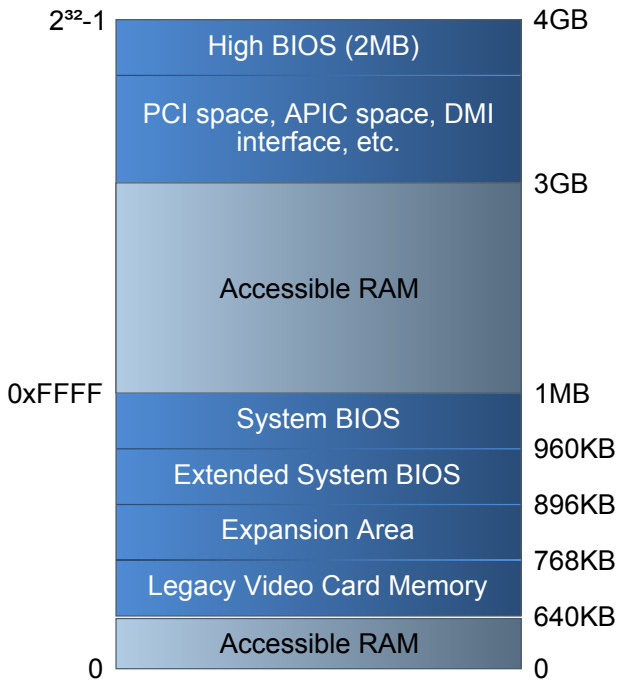


Abb. 15: Physikalischer Speicher eines Intel PC (Duarte [2008])

Physikalischer Speicher und Betriebsmodi

Abb. 15 zeigt den typischen Aufbau des physikalischen Speichers eines Intel-PCs mit 4 GB physikalischem Adressraum nach der Initialisierung durch das System-BIOS (*Basic Input Output System*) beim Starten des Rechners.

Nicht nur der eigentliche Hauptspeicher, das RAM (*Random Access Memory*), liegt im physikalischen Adressraum, sondern auch der I/O-Bereich (*Input/Output*).

Über die Adressen des I/O-Bereichs werden Peripherie, PCI-Bus, Grafikkarte usw. angesprochen. Auch wenn in einem solchen PC 4 GB RAM eingebaut sind, so wird ein Teil davon durch BIOS und I/O überlagert und ist somit nicht nutzbar.

Bei den ersten PCs konnte nur auf die ersten 640 KB zugegriffen werden, und dieser Zugriff war für alle Anwendungen uneingeschränkt. Dies lag daran, dass die Prozessoren (8086) einen kleineren Adressbus hatten und weder über Privilegierungen noch über eine Speicherverwaltung verfügten. Sie liefen im sogenannten *Real Mode* (s.u.).

Die Nachteile und Einschränkungen eines solchen Szenarios sind klar:

1. Das genaue Speicher-Layout unterscheidet sich je nach verwendeter Hardware. Fest im Programmcode definierte physikalische Adressen führen zu Kompatibilitätsproblemen.
2. Für Multitasking muss garantiert sein, dass verschiedene gleichzeitig laufende Programme auch unterschiedliche Speicherbereiche nutzen. Falls im Programmcode aber statische physikalische Adressen angegeben sind, ist die Wahl des Speicherbereichs nicht mehr variabel. Das Betriebssystem kann nicht sicherstellen, dass sich die Bereiche nicht überlappen.
3. Mit physikalischen Adressen spricht man direkt das RAM und andere Komponenten an. Somit ist es nicht möglich, bestimmte Speicherbereiche gegen unprivilegierten Zugriff zu schützen.

Betriebsmodi

Die x86-Prozessorfamilie kennt fünf verschiedene Betriebsmodi, die sich im Laufe der Zeit entwickelt haben:

1. **Real Mode:** Der Real Mode ist der einzige Modus, den schon die 8086-CPU kannte. Die Adressierungsbreite beträgt 20 Bit, womit 1 MByte physikalisch adressierbar ist. (Auch moderne PCs arbeiten beim Hochfahren zunächst im Real Mode, wechseln dann aber in den Protected Mode, den moderne Betriebssysteme nutzen.)
   Die Segmentregister waren bereits beim 8086 vorhanden. Die Berechnung einer 20-Bit-physikalischen Adresse aus 16-Bit-Segmentregister und 16-Bit-logischer Adresse (Offset) geschieht folgendermaßen: Die Segmentadresse wird mit 16 multipliziert (um vier Bit nach links geshiftet), und der Offset wird addiert.

B

Beispiel 17

Aus Segmentadresse und Offset

```
Segment: D437h bzw.  1101010000110111
Offset:  0158h bzw.  0000000101011000
```

berechnet sich die physikalische Adresse wie folgt:

```
Physikalische Adr.   = Segment*10h + Offset
                     = D4370h + Offset
                     = D44C8h bzw. 11010100010011001000
```

   Programme im Real Mode haben uneingeschränkten Zugriff auf Speicher und I/O. Es gibt keinerlei Schutzmaßnahmen. MS-DOS ist ein Beispiel für ein Betriebssystem, das den Real Mode verwendet.

Protected Mode

2. **Protected Mode:** Der Protected Mode wurde mit der 80286-CPU eingeführt. Er erlaubt den Zugriff auf den Bereich oberhalb 1 MB physikalischen Adress-

raums und verwendet Segmentierung. Er bringt außerdem den linearen und logischen Adressraum und das Ring-Privilegiensystem mit sich. So lassen sich die Eigenschaften eines Mehrbenutzerbetriebssystems, also die quasi gleichzeitige Ausführung mehrerer Programme sowie System-, Speicher- sowie Ein-/Ausgabe-Schutz realisieren. Die Adressierung erfolgt mittels einer „Beschreibungstabelle“ (*Description Table*), die die jeweiligen Informationen über die Segmente enthält. Dies wird im folgenden Abschnitt beschrieben.

3. **Long Mode:** Der Long Mode ist ein Betriebsmodus der 64-Bit x86-64-Architektur und erlaubt die Verwendung der 64-Bit-Register. Er ist für 64-Bit-Applikationen bestimmt. 16- und 32-Bit-Applikationen hingegen werden im „Kompatibilitätsmodus“ bzw. Protected Mode ausgeführt.
4. **Virtual Mode:** Im Virtual Mode werden Programme ausgeführt, die nicht zum laufenden Protected Mode kompatibel sind. Die 20-Bit-Segmente des Real Mode werden im Speicher simuliert.
5. **System Management Mode:** Der System Management Mode beschäftigt sich weniger mit der Ausführung und Speicherverwaltung von Programmen im Allgemeinen. Er wurde mit der Einführung der Pentium-Prozessoren Anfang der 1990er Jahre zum ersten Mal vorgestellt. In diesem Modus werden laufende Befehle/Programme zeitweilig gesperrt, um höher privilegierte Prozesse oder Stromsparmodi ausführen zu können.

Im Rahmen dieses Moduls werden wir uns fast ausschließlich mit dem Protected Mode beschäftigen.

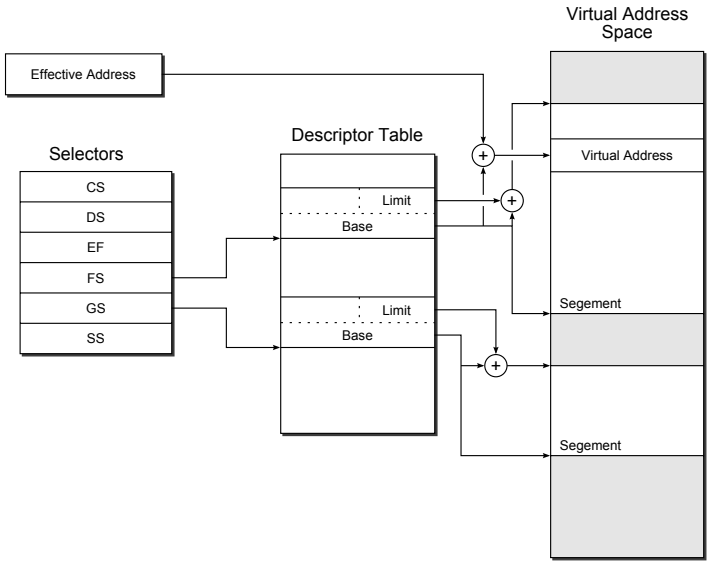
Segmentierung

Segmentregister

Im Protected Mode werden weiterhin die bekannten Segmentregister (`cs,ds,ss,es` usw.) verwendet, jedoch in einer anderen Weise als im Real Mode. Die Segmentregister enthalten nun einen Zeiger auf einen Eintrag in der *Segment Description Table* (SDT) sowie die Privilegstufe des Programms, zu dem sie gehören. Ein Eintrag in der SDT enthält die Basisadresse des Segments im linearen Adressraum. Zudem enthält der Eintrag ein Limit, das angibt, wie groß das Segment ist und eine Privilegvorgabe, die angibt, welche Privilegstufe ein Programm mindestens haben muß, um auf das Speichersegment zugreifen zu dürfen. Zur Berechnung der linearen Adresse werden logische Adresse (effektive Adresse) und Basisadresse addiert (Abb. 16). Liegt die berechnete logische Adresse oberhalb des Limits oder passen die Privilegstufen nicht zueinander, so wird ein Fehler ausgelöst, eine sogenannte Exception.

Die Segmente können prinzipiell eine beliebige Länge haben und an einer beliebigen Stelle im Speicher platziert sein. Die Verwaltung der SDT übernimmt das

Abb. 16: Segmentierung im Protected Mode (Intel [2012])



Betriebssystem. Dieses teilt einem Programm beim Start seine Segmente zu und füllt die SDT entsprechend.

B

Beispiel 18

Das Beispiel zeigt den Zugriff auf die logische Adresse 1234 im Segment `fs`. Segmente können in Assembler explizit angegeben werden.

```
mov eax, fs:[1234]
```

Zur Umwandlung der logischen Adresse 1234 in die entsprechende lineare Adresse benötigt die Memory Management Unit (MMU) Informationen aus der Segment Description Table. `fs` zeigt auf einen Eintrag in der SDT. Dort sind Basisadresse und Limit des Segments hinterlegt. Zunächst prüft die MMU, ob die logische Adresse über dem Limit liegt und gibt ggf. einen Fehler aus. Liegt die logische Adresse innerhalb des Segments, wird sie mit der Basisadresse addiert. Das Ergebnis ist die gewünschte lineare Adresse.

Wie im Beispiel beschrieben können Segmente bei Speicherzugriffen explizit ausgewählt werden. Solche expliziten Adressierungsvorgaben werden *Far Pointer* genannt. Der Regelfall sind *Near Pointer* ohne Segmentangabe. Die Wahl des Segments entscheidet sich dann aus dem Kontext, wie es in den folgenden Beispielen dargestellt ist:

- `jmp 1234`
  Befehle, die sich auf den Programmcode beziehen, verwenden immer das cs-Segment.
- `mov eax, [1234]`
  Datenreferenzen beziehen sich immer auf das ds-Segment.
- `push eax`
  Stack-Operationen verwenden immer das ss-Segment.

K

Kontrollaufgabe 7

Welche Absicht könnte hinter so einem Befehl stecken?

```
mov cs:[1234], 5430h
```

Die Segmentierung erlaubt es einem Multitasking-Betriebssystem, die Adressbereiche von gleichzeitig ablaufenden Prozessen voneinander zu isolieren. Bei einem Prozesswechsel müssen dazu lediglich die Segmentregister neu geladen werden. Insbesondere durch die Prüfung der Limitüberschreitung ist es nicht mehr möglich, auf Adressbereiche anderer Prozesse zuzugreifen.

Vorteile der Segmentierung sind die

- feingranulare Rechteverwaltung,
- die Möglichkeit des Adressraumwechsels und
- der Adressraumschutz.

Allerdings erschwert Segmentierung die Auslagerung des Speichers auf eine Festplatte, weil nur komplette Segmente ausgelagert werden können. Von modernen Betriebssystemen wird die Segmentierung deshalb nicht mehr verwendet. Im Protected Mode kann sie zwar nicht abgeschaltet werden, aber man kann sie transparent schalten, indem alle Segmente an der Startadresse 0 anfangen und die maximale Länge $2^{32} - 1$ bekommen. In diesem Fall sind logische und lineare Adresse identisch. Zur Programmanalyse, insbesondere zur Analyse älterer Programme auf älteren Systemen, ist es allerdings erforderlich, mit der Technik der Segmentierung vertraut zu sein.

Statt Segmentierung wird heutzutage Paging eingesetzt.

Paging

MMU

Beim Paging wird der lineare Speicher nicht in Segmente, sondern in Seiten (*Pages*) gleicher Größe unterteilt. Diese Pages sind entweder 4 KB oder 4 MB groß. Den Pages werden von der MMU *Page Frames* im physikalischen Speicher zugeordnet. Page Frames haben immer dieselbe Größe wie Pages, können aber beliebig angeordnet sein. Dies ist in Abbildung 17 dargestellt.

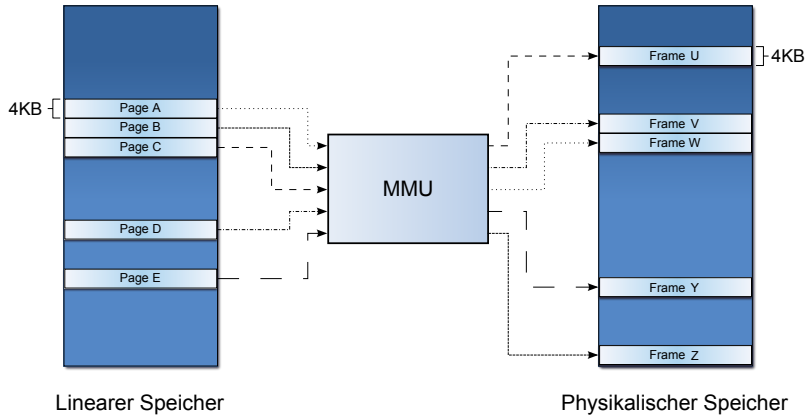
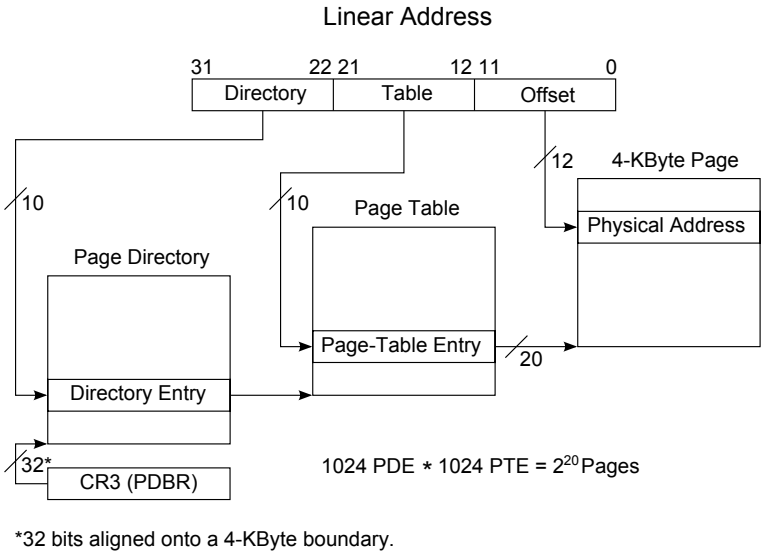


Abb. 17: MMU: Pages und Frames (Freiling et al. [2010])

Page Table

Beim Paging erfolgt die Übersetzung von linearen in physikalische Adressen in der MMU mit Hilfe von *Page (Translation) Tables*. Die lineare Adresse wird zunächst (logisch) in Directory Index, Table Index und Offset aufgespalten. Für eine Seitengröße von 4 KB ist dies in Abbildung 18 dargestellt. Der Directory Index umfasst 10 Bit, somit hat das Page Directory 1024 Einträge.

Abb. 18: Paging (80386) bei 4 KB großen Pages (Intel [2012])



Das Verfahren zur Ermittlung einer physikalischen Adresse ist zweistufig:

Page Directory

1. Zuerst wird im Page Directory mit Hilfe des Directory Index die Adresse der Page Table ausgelesen. Der Table Index gibt den Offset in der gewählten Page Table an.
2. Eine Page Table hat wiederum 1024 Einträge, und der gewünschte Eintrag wird durch den Table Index ausgewählt. Aus der Page Table wird schließlich die Basisadresse des zugehörigen Page Frame ausgelesen. Darauf wird der Offset addiert, was die entsprechende physikalische Adresse generiert.

Bei einer Seitengröße von 4 MB fällt die Page Table weg, und die untersten 22 Bit der logischen Adresse geben den Offset im Page Frame an.

Wie können nun die Adressräume von Prozessen getrennt werden?

schneller Wechsel des Adressraums

Jeder Prozess erhält vom Betriebssystem beim Start sein eigenes Page Directory und ggf. seine eigenen Page Tables. Die Steuerung erfolgt über die Control Register `CR0` bis `CR4`, die ab dem 80386 existieren. Neben diversen Status-Flags und Options Flags zur Speicherverwaltung in `CR0` und `CR4`, die hier nicht weiter betrachtet werden sollen, sind `CR2` und `CR3` wesentlich. `CR3` zeigt auf das Page Directory des aktuellen Prozesses. Ein Adressraumwechsel ist also ganz einfach durch Anpassung dieses Registers möglich. Im Fall eines Seitenfehlers enthält `CR2` die virtuelle Adresse, auf die hätte zugegriffen werden sollen.[11]

Neben den Adressen sind in den Einträgen des Page Directory und der Page Table noch weitere Informationen gespeichert. Unter anderem sind die folgenden Attribute und Flags dort eingetragen:

[11] Den Inhalt von CR2 hat jeder Windows-Benutzer mit hoher Wahrscheinlichkeit schon einmal in der Form „Allgemeine Schutzverletzung bei...“ auf seinem Bildschirm gesehen. :-)

- Das User/Supervisor Flag kann den Zugriff auf eine Seite für Ring 0 beschränken.
- Das Read/Write-Attribut schränkt den Schreibzugriff ein. Hierdurch kann insbesondere eine Page in den Read-Only-Modus versetzt werden.
- Das Present Flag gibt an, ob ein Page Frame bzw. eine Page Table auf die Festplatte ausgelagert ist, oder ob sie sich im Hauptspeicher befinden.
- Das PageSize Flag definiert, ob die Seitengröße 4 KB oder 4 MB beträgt. Bei einer Seitengröße von 4 MB wird auf eine Page Table verzichtet, und der Offset ist statt 12 Bit 22 Bit lang.

Insgesamt ist jeder Tabelleneintrag 32 Bit breit. Auf den genauen Aufbau wollen wir hier jedoch nicht eingehen.

Abb. 19 zeigt noch einmal zusammengefasst die hintereinander geschalteten Funktionsblöcke von Segmentierung und Paging von IA-32 mit der Umwandlung einer logischen in eine physikalische Adresse.

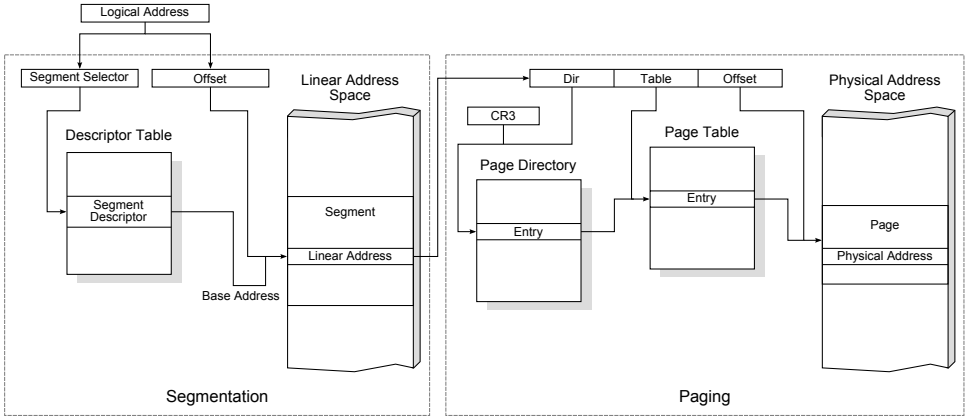


Abb. 19: Adress- Umwandlung mit Segmentierung UND Paging (Intel [2012])

> Kontrollaufgabe 8
>
> Wie viele Pages kann eine 80386-CPU verwalten, wenn eine Page-Größe von 4 KB bzw. 4 MB vorgegeben ist?

K

4.8 String-Befehle

Nach einem längeren Ausflug in die Welt der Speicherverwaltung wird nun der Befehlssatz von IA-32 vervollständigt.

Die bekanntesten CISC-Befehle bei den x86-Prozessoren sind die String-Befehle `movs` (Move String), `cmps` (Compare String), `scas` (Scan String), `lods` (Load String), und `stos` (Store String). Diese Befehle wurden speziell zur Beschleunigung von häufig benötigten C-Standardfunktionen wie `memcpy()`, `memset()`, `strlen()`, `strcmp()` etc. entwickelt.

String-Befehl

Die String-Befehle erlauben es, große Datenblöcke, wie z. B. alphabetische Zeichenketten oder Zahlenlisten, direkt im Speicher zu verschieben oder zu bearbeiten, ohne den Umweg über ein CPU-Register gehen zu müssen. Diese Befehle operieren mit den individuellen Elementen eines String, d. h. eines eindimensionalen Feldes von Bytes, Worten oder Doppelworten. Die zu bearbeitenden String-Elemente werden über das `esi`-Register (Source Index) und das `edi`-Register (Destination Index)

referenziert. Beide Register enthalten effektive Adressen, die auf die aktuellen String-Elemente zeigen.[12]

Allen String-Befehlen ist die implizite Verwendung der Register `esi` und/oder `edi` zur Adressierung, oft verknüpft mit dem Register `eax` gemeinsam. Die beteiligten Indexregister `esi` und `edi` werden nach der Ausführung automatisch um die Operandengröße in Byte (1, 2 oder 4) inkrementiert (bei Directionflag DF=0) bzw. dekrementiert (bei Directionflag DF=1).

`rep`-Präfix

Alle String-Befehle können mit dem Wiederholungspräfix `rep` (*repeat*) versehen werden. Dies dient zur Implementierung von Zählschleifen. Das `rep`-Präfix benutzt implizit das `ecx`-Register als Schleifenzähler. Die Wiederholungspräfixe `repe` (*repeat while equal*) und `repne` (*repeat while not equal*) prüfen zusätzlich als zweites Endekriterium das Zeroflag.

Wir betrachten nun als Beispiel den `movs`-Befehl etwas genauer; für die übrigen String-Befehle gilt das Gesagte analog. `movs` kopiert ein Byte, Wort oder Doppelwort von der mit `[esi]` als Quelle adressierten Speicherstelle an die mit `[edi]` als Ziel bestimmte Speicherstelle. Beim allgemeinen `movs`-Befehl muss die Operandengröße durch Typoperatoren (`doubleword, word` oder `byte`) explizit angegeben werden.

Der Einfachheit halber existieren Kurzformen des `movs`-Befehls ohne explizite Angabe der Operandengröße. `movsb` (*move string of bytes*), `movsw` (*move string of words*) und `movsd` (*move string of doublewords*) geben die Operandengröße implizit an. (Bei den übrigen String-Befehlen ist dies analog der Fall.) Nach der Kopieroperation werden die Indexregister `esi` und `edi` automatisch um die Operandengröße (also 1, 2 oder 4) inkrementiert (bei Directionflag DF=0) bzw. dekrementiert (Directionflag DF=1).

`movs` kann mit dem `rep`-Präfix kombiniert werden. Das `rep`-Präfix dekrementiert den Inhalt des `ecx`-Registers und wiederholt den `movs`-Befehl solange, bis das `ecx`-Register gleich 0 ist.

B

Beispiel 19

Die C-Funktion

```
void* memcpy(void* dst, const void* src, size_t count);
// kopiere "count" Bytes von src nach dst
```

kann folgendermaßen in Assembler implementiert werden:

```
cld              ; clear direction flag => increment esi, edi
mov ecx, [count] ; init ecx for rep
mov esi, [src]   ; init esi for movsb
mov edi, [dst]   ; init edi for movsb
rep movsb
```

`cld` initialisiert das Directionflag. Nach der Initialisierung von Zähler, Quelle und Ziel wird der Kopierbefehl so oft wiederholt, bis `ecx` = 0 ist.

[12] Der aufmerksame Leser wird hier sicher sofort den Einwand vorbringen, dass selbst bei einem `movs`-Befehl mit nur einem Byte gleichzeitig ein lesender und ein schreibender Zugriff auf unterschiedliche Adressen des Hauptspeichers erfolgt, was IA-32 eigentlich nicht erlaubt. In Wahrheit wird der CISC-Kopierbefehl CPU-intern durch Mikrocode in getrennte Befehle aufgespalten, die nacheinander ausgeführt werden.

In Tabelle 12 sind die wichtigsten String-Befehle in der Doppelwortversion und die drei möglichen Präfixe dargestellt. Bei den 16-Bit- bzw. 8-Bit-Varianten ist ggf. das Register `eax` durch `ax` bzw. `al` zu ersetzten, ansonsten bleibt die Funktionalität die gleiche.

Tabelle 12: StringBefehle und Präfixe

Befehl	Beschreibung
movsd	kopiert 32-Bit-Doppelwort von [esi] nach [edi]
stosd	kopiert eax nach [edi]
lodsd	kopiert [esi] nach eax
cmpsd	vergleicht [esi] mit [edi] und setzt die Status-Flags
scasd	vergleicht eax mit [edi] und setzt die Status-Flags
rep	wiederholt String-Befehl, bis ecx=0
repe	(repeat while equal) wiederholt String-Befehl, bis ecx=0 oder ZF=0
repne	(repeat while not equal) wiederholt String-Befehl, bis ecx=0 oder ZF=1

Kontrollaufgabe 9

In Zusammenhang mit welchen String-Befehlen machen die Präfixe `repe` und `repne` Sinn? Welche Funktionalität ließe sich damit bspw. leicht implementieren?

K

4.9 Sonstige Befehle

In diesem Anschnitt werden einige weitere Befehle erklärt, die die Vorstellung des Befehlssatzes von IA-32 abrunden (Tabelle 13).

Anmerkungen:

- Einen Befehl der ersten Gruppe haben wir bereits im vorangegangenen Beispiel bei den String-Befehlen kennengelernt. Die Flag-Befehle dienen zur direkten Manipulation von Flags. Damit können einzelne Flags gezielt gesetzt oder gelöscht werden.
- Der Befehl `nop` (*no operation*) erhöht lediglich den `eip`, ansonsten passiert nichts.
- Die Schleifenbefehle sind typische Vertreter aus dem Repertoire der CISC-Befehle. Sie dekrementieren das `ecx`-Register und führen einen Sprung aus, wenn `ecx` ungleich 0 ist. `loope` bzw. `loopne` überprüfen zudem das Zeroflag. Die Schleifenbefehle werden zur kompakten Implementierung von Zählschleifen verwendet. Zu beachten ist, dass die Zieladresse nur 128 Byte vom jeweiligen `loop`-Befehl entfernt sein darf.
- Die Port-Befehle lesen/schreiben Werte von/in einen I/O-Port. Die I/O-Ports sind in der Regel Peripheriegeräten zugeordnet wie z. B. dem Tastaturpuffer. Die direkte Ausführung von Port-Befehlen ist normalerweise dem Betriebssystem vorbehalten. Portadressen kleiner `100h` können direkt angegeben werden. Höhere Portadressen müssen über das `dx`-Register benannt werden.
- Die Konvertierungsbefehle erweitern vorzeichenbehaftete Zahlen von 8 auf 16, von 16 auf 32 bzw. von 32 auf 64 Bit.
- Der Befehl `setz` steht hier stellvertretend für eine ganze Reihe von Befehlen zum Setzen eines Byte. Der Operand ist ein 8-Bit-Register oder eine Byte-Variable im Speicher. In Abhängigkeit vom Flag-Status setzt der Befehl den Operanden auf 0 oder 1.

Tabelle 13: Diverse Befehle

Befehl	Beschreibung
stc	setzt das Carryflag (CF)
clc	löscht CF
cmc	invertiert CF
std	setzt das Directionflag (DF) (siehe String-Befehle)
cld	löscht DF
sti	setzt das Interruptflag (IF)
cli	löscht IF
nop	keine Aktion
loop adr	Dekrementiert ecx; Sprung nach adr, falls ecx <> 0
loope adr	Dekrementiert ecx; Sprung nach adr, falls ecx <> 0 und ZF=1
loopne adr	Dekrementiert ecx; Sprung nach adr, falls ecx <> 0 und ZF=0
loopz adr	wie loope
loopnz	wie loopne
in dest, port	liest einen Wert von einem externen Port
out port, source	schreibt einen Wert in einen externen Port
cbw	ax=al (mit Vorzeichen) konvertiert byte zu word
cwd	dx:ax=ax (mit Vorzeichen) konvertiert word zu double word
cwde	eax=ax (mit Vorzeichen) konvertiert word zu double word
cdq	edx:eax=eax (mit Vorzeichen) konvertiert double word zu quad word
setz op	Operand = 1, falls ZF=1, sonst 0
int op	
iret	

- Die Interrupt-Befehle `int` und `iret` sind ähnlich zu den Unterprogrammbefehlen `call` und `ret`. Interrupts rufen spezielle Unterprogramme auf, die *Interrupt oder Exception Handler*. Interrupts werden im folgenden Abschnitt genauer behandelt.

Neben diesen Befehlen existiert noch eine große Anzahl weiterer Befehle, wie z. B. die Protected-Mode-Befehle. Diese steuern insbesondere Paging und Segmentierung, weshalb sie für den Anwendungsprogrammierer nicht zugänglich sind. Sie werden bei der Implementierung von Betriebssystemen gebraucht.

4.10 Interrupts und Exceptions

Interrupt

Interrupts (Unterbrechungen) unterbrechen den „normalen“ Programmablauf. Sie rufen bestimmte Unterprogramme auf, die *Interrupt Handler* genannt werden. Prinzipiell unterscheiden sich die Aufrufe dieser Interrupt Handler nicht von den Aufrufen anderer Unterprogramme.

Interrupts können auf unterschiedliche Weise ausgelöst werden.

1. Manche Interrupts löst der Prozessor selbst aus, wenn die Ausführung eines Befehls fehlerhaft war. Typische Beispiele hierfür sind eine Division durch 0, oder eine Schutzverletzung beim Speicherzugriff. Die Aufgabe des Interrupt Handler besteht darin, das Problem zu lösen oder wenigstens eine Fehlermeldung auf dem Bildschirm zu generieren. Da diese speziellen

Interrupts eigentlich nur in Ausnahmefällen auftreten sollten, werden sie auch *Exceptions* genannt.

2. Interrupts können von externen Quellen ausgelöst werden, bspw. durch Eingabegeräte wie Tastatur und Maus. In diesen Fällen ist es oft sinnvoll, das aktuell laufende Programm zu unterbrechen, um z. B. den Mauszeiger auf dem Bildschirm zu aktualisieren.[13] Neben diesen asynchronen Interrupts gibt es auch externe Interrupts, die bspw. durch Timing-Bausteine ausgelöst werden. Solche Timer Interrupts werden benutzt, um einen Prozesswechsel bei einem Multitasking-Betriebssystem einzuleiten. Weitere Beispiele für externe Interrupts sind DMA(*Direct Memory Access*)-Anforderungen von Festplatten und Netzwerkkarten, um großvolumige Datentransfers bevorzugt abzuarbeiten.
3. Interrupts können durch ein Programm explizit mit Hilfe des `int`-Befehls ausgelöst werden. Statt einer Unterprogrammadresse wird hierbei lediglich eine Interrupt-Nummer angegeben. Solche Interrupts werden für Betriebssystemaufrufe genutzt.

Interrupt Vector Table

Die zentrale Datenstruktur zur Verwaltung von Interrupts ist die IVT (*Interrupt Vector Table*). In dieser werden die Adressen der Interrupt Handler hinterlegt. Der Zugriffsindex ist die Interrupt-Nummer. Abb. 20 zeigt den Aufbau der IVT für IA-32. Die Nummern sind für die Exceptions fest vorgegeben, die weiteren werden durch die Hardware des Rechners oder das Betriebssystem bestimmt oder sind frei nutzbar. Die IVT selbst liegt im Hauptspeicher. Bei einem PC im Protected Mode liegt sie bspw. ab der physikalischen Adresse 0 an aufwärts.

Vektor	Beschreibung
0	Integer Divide-by-Zero Exception
1	Debug Exception
2	Non-Maskable-Interrupt
3	Breakpoint Exception (INT 3)
4	Overflow Exception (INTO instruction)
5	Bound-Range Exception (BOUND instruction)
6	Invalid-Opcode Exception
7	Device-Not-Available Exception
	...
13	General-Protection Exception
14	Page-Fault Exception
	...
20-255	Nicht belegt, frei verfügbar durch Betriebssystem

Abb. 20: Aufbau der Interrupt Vector Table

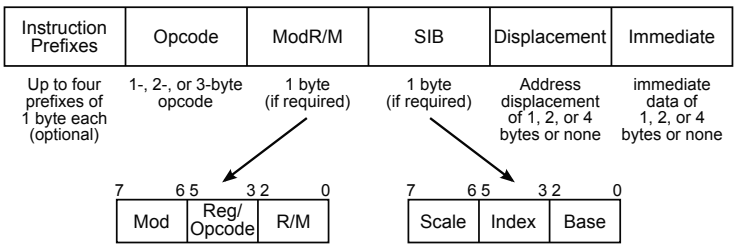
4.11 Maschinenbefehlsformat

Wie bei CISC-Architekturen üblich verfügt IA-32 über kein einheitliches Befehlsformat. Die Maschinenbefehle haben hier eine variable Länge von 1 bis zu 15 Byte. Beispielsweise wird der `ret`-Befehl zu `C3h` übersetzt und ist damit nur 1 Byte lang, `sub ebp,[esp+ebx*2]` hingegen benötigt 3 Byte: `2B2C5Ch`.

Die einzelnen Befehle haben bis zu sieben Bestandteile (s. Abb. 21).

[13] Um beim Beispiel der Maus zu bleiben: Der Zeitpunkt der Mausbewegung ist nicht voraussagbar; es handelt sich um ein asynchrones Ereignis. Die Alternative zu einem solchen asynchronen Interrupt könnte nur darin bestehen, permanent den Mauszustand abzufragen, was man auch als *Polling* bezeichnet.

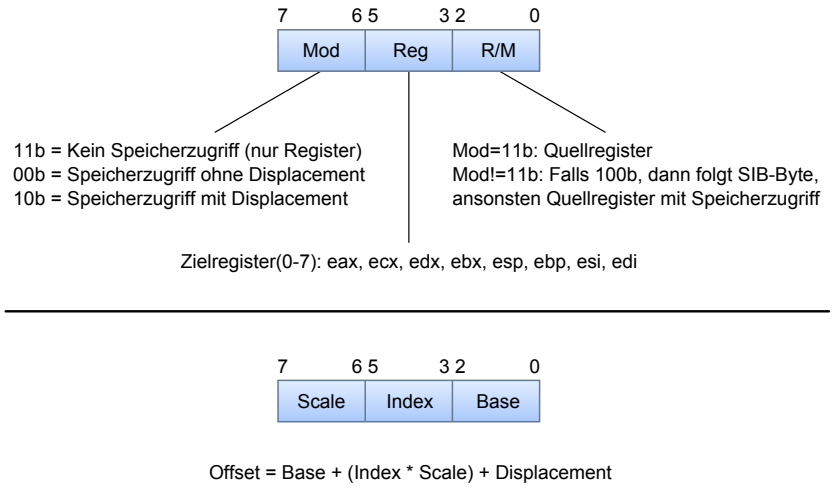
Abb. 21: IA-32-Befehlsformat (Intel [2012])



Die Komponenten haben folgende Bedeutung:

- Der Opcode gibt den Befehlstyp an , z. B. `add, push, call` oder `mov`.
- Die Immediate Bytes sind für Konstanten vorgesehen.
- Displacement-Werte werden bspw. bei der Offset-Adressierung benutzt.
- Es existieren vier verschiedene Gruppen von Präfixen; jeder Befehl darf ein Präfix jeder Gruppe benutzen. Die erste Gruppe beinhaltet das optionale `rep`-Präfix für String-Befehle. Die übrigen werden dazu benutzt, die standardmäßige Adressgröße, Operandengröße und das Standardsegment eines Befehls zu ändern.
- ModRM beschreibt die verwendeten Operanden (Register und/oder Speicher) und die Adressierungsart.
- Das SIB-Byte gibt Scale, Index und Base bei der Offset-Adressierung an. Abb. 22 zeigt den genauen Aufbau von ModRM und SIB.

Abb. 22: ModRM und SIB (Freiling et al. [2010])



Die Maschinenbefehle werden nach der Ordnung Little Endian im Speicher abgelegt, beginnend mit den Präfix-Bytes auf den niedrigsten Speicheradressen.

5 Zusammenfassung

Auf IA-32 basiert ein großer Teil der heute im Gebrauch befindlichen Computer. Diese Architektur bietet sich daher besonders zur Erlernung der Assemblerprogrammierung an.

Für die Erstellung von Assemblerprogrammen benötigt man ein Programmiermodell. Dieses ergibt sich natürlich aus dem Aufbau einer Rechnerarchitektur.

Diesen Aufbau muss man grob verstehen, wobei man allerdings die einzelnen Funktionseinheiten meist als „black boxes“ betrachten kann, deren Inhalte für den Programmierer im Detail nicht wichtig sind. Für den Programmierer entscheidend sind die Register und der Befehlssatz einer Architektur. Darüber hinaus muss der Programmierer wissen, wie die Kommunikation mit dem Umfeld, insbesondere mit dem Hauptspeicher funktioniert. Kenntnisse über die Mechanismen von Adressierung und Speicherverwaltung sind gerade bei einer komplexen Architektur wie IA-32 sehr wichtig.

Die Speicherverwaltung ist eine der Besonderheiten von IA-32, und ihre Realisierung ist eine individuelle Lösung für diese Architektur. Dies trifft ebenso für Interrupts bzw. Exceptions zu. Bei einer anderen Architektur mögen Interrupts ganz anders realisiert sein als bei IA-32.

6 Übungen

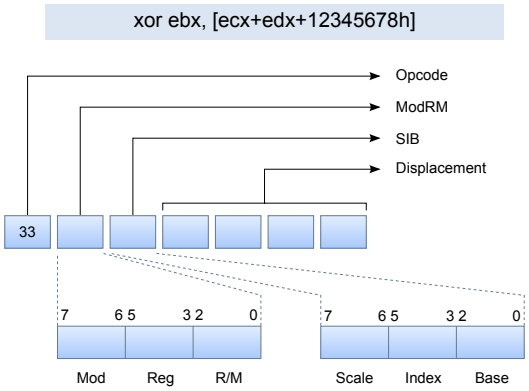
Ü

Übung 1

Im letzten Abschnitt dieses Mikromoduls wurde das Maschinenbefehlsformat von IA-32 vorgestellt. Ergänzen Sie für den angegebenen Befehl die binären Werte in den Kästchen in Abb. 23. Die Angaben in den Abbildungen 21 und 22 reichen dazu aus.

Hilfestellung: *Scale* kann vier verschiedene Werte haben: 1, 2, 4 oder 8. Diese sind in 2 Bit in der Form 00, 01, 10 bzw. 11 codiert. Die in Abb. 22 angegebenen 8 Register sind in der angegebenen Reihenfolge in 3 Bit codiert, 110 steht bspw. für `esi`.

Abb. 23: Befehlsformat



Allgemeiner Hinweis zu den folgenden Übungen: In einigen Teilen der folgenden Übungsaufgaben werden rudimentäre Kenntnisse in C oder C++ vorausgesetzt. Sollten Sie nicht über diese Kenntnisse verfügen, so können Sie diese Teile der Übungsaufgaben überspringen. Im Internet existiert aber auch eine Fülle von Kurzanleitungen zu C; hiermit können Sie in kurzer Zeit die Wissenslücken schließen.

Ü

Übung 2

In dieser Übung sollen Sie sich mit den Grundlagen der Programmierung in Intel-Assembler vertraut machen. Dafür wird der Microsoft Macro Assembler (MASM), genauer gesagt das MASM32-Paket, verwendet. Dieses ist im Modulmaterial auf der Lernplattform enthalten. Darin befinden sich auch zwei simple Beispiele, die als Template verwendet werden können sowie eine Batch-Datei zum Aufrufen des Assemblers. Falls Sie MASM32 in ein anderes Verzeichnis als das Standardverzeichnis installieren, dann muss der entsprechende Pfad in der Batch-Datei angepasst werden. Der erste Parameter ist die Eingabedatei (`.asm`), der zweite Parameter ist das sogenannte Subsystem. Für Konsolenanwendungen muss dies `CONSOLE` sein, ansonsten `WINDOWS`. Sie können aber auch die im MASM-Paket enthaltene Bedienoberfläche *qeditor.exe* benutzen, die einem die „Handarbeit“ abnimmt, aber etwas gewöhnungsbedürftig ist.

Schreiben Sie ein Konsolenprogramm in Assembler, das zwei ganze Zahlen als Benutzereingabe einliest und die Summe beider Zahlen ausgibt.

Hinweis: Sie können für die Interaktion mit der Konsole Funktionen aus der C-Standardbibliothek verwenden. Im Internet existieren dafür diverse Dokumentationen. Es bietet sich an, `printf` zur Textausgabe und `scanf` zur Eingabe zu verwenden. In MASM muss jeder Standard-C-Funktion ein `crt_` vorangestellt werden (also z. B. `crt_printf`, `crt_scanf` etc.).

Übung 3

Ü

Als nächstes sollen Sie sich mit bedingten Verzweigungen vertraut machen. Schreiben Sie dafür ein Konsolenprogramm in Assembler, das eine beliebige Benutzereingabe – vorzugsweise eine ganze Zahl – einliest und in Abhängigkeit davon zwei verschiedene Ausgaben erzeugt.

Übung 4

Ü

Es sind folgende x86-Instruktionen gegeben:

```
mov eax, 100
mov [eax], DEADCODEh
```

Wie lauten die Speicherwerte an den Adressen 100, 101, 102 und 103, nachdem diese beiden Befehle ausgeführt wurden?

Übung 5

Ü

Angenommen, es stehen die folgenden Variablendeklarationen in einer Hochsprache (z. B. C oder Java) zur Verfügung (*int* steht für eine 32 Bit breite Variable, `char` für eine 8 Bit breite Variable):

```
(1) int x;
(2) int i;
(3) int arr[256];
(4) char ch[8];
```

Gehen sie weiterhin davon aus, dass die Speicheradressen zu den Variablen `x`, `arr`, `ch` und `i` jeweils bereits in die Register `esi`, `edi`, `ebx` und `ecx` geladen sind. Übersetzen Sie die folgenden Hochsprachenausdrücke, wie bei den ersten zwei Beispielen bereits vollständig geschehen, in jeweils einen x86-Befehl. Das Ergebnis soll mit Ausnahme von Zeile 4 immer in das Register `eax` geladen werden. In Zeile 4 wird ein Byte adressiert, sodass das Ergebnis in das Register `al` geladen werden soll.

Ausdruck	**x86-Befehl**
`123`	`mov eax, 123`
`x`	`mov eax, [esi]`
`arr[13]`	`mov eax, ...`
`ch[i]`	
`arr[i + 13]`	

Ü

Übung 6

Beim Analysieren einer Binärdatei stoßen Sie auf folgenden Unterprogrammaufruf:

Instruktionen	Kommentar
`push eax`	
`push ecx`	
`call CAFEBABEh`	
`...`	

An der Speicherstelle `CAFEBABEh` befinden sich die in der folgenden Tabelle aufgeführten Instruktionen:

Instruktionen	Kommentar
`push ebp`	Wert von `ebp` wird auf Stack gesichert.
`mov ebp, esp`	
`sub esp, 4`	
`mov ecx, [ebp+8]`	
`add ecx, [ebp+12]`	
`mov [ebp-4], ecx`	
`inc [ebp-4]`	
`mov eax, [ebp-4]`	
`mov esp, ebp`	
`pop ebp`	
`ret 8`	

a) Geben Sie zu jeder Instruktion in dem entsprechenden Kommentarfeld an, welchen Zweck diese hat.

b) Was macht die Subroutine genau? Wie könnte die entsprechende Subroutine in einer Hochsprache aussehen?

c) Welche Instruktionen in obiger Tabelle zählen zum Prolog, welche zum Epilog?

Stichwörter

`rep`-Prafix, 44

Adressierungsarten, 24
arithmetische Befehle, 16

bedingter Sprung, 19

Datentransferbefehle, 16

effektive Adresse, 25
Endianness, 23
Exception, 47
Exception Handler, 46
Explizite Interrupts, 47
Externe Interrupts, 47

Flags, 11

IA-32, 7
Immediate-Adressierung, 24
indirekte Adressierung, 19
Intel Reference Manuals, 7
Intel Syntax, 14
Interrupt, 46
Interrupt Vector Table, 47

Kontrolltransferbefehle, 18

Linearer Adressraum, 36
Logischer Adressraum, 36

Malware, 8
Maskierung, 23
Memory-Adressierung, 25
Mikrocode, 44
MMU, 37, 41

Offset-Adressierung, 25

Page Directory, 42
Page Table, 42
Pages und Frames, 41
Parameter, 30
Physikalischer Adressraum, 36
Privilegiensystem, 35
Prolog und Epilog, 31
Protected Mode, 38

Real Mode, 38
Registeradressierung, 25
Registersatz, 8
Registersatz 80386, 9
Rotationsbefehl, 17

Shift-Befehl, 17
Stack, 27
Stack Frame, 30
Stack-Register, 10
Status-Flag, 11
String-Befehl, 43
System-Flag, 12

unbedingter Sprung, 19
Unterprogramm, 29

x86-Prozessorfamilie, 7

Liste der Lösungen zu den Kontrollaufgaben

Lösung zu Kontrollaufgabe 1 auf Seite 18

Teil 1:

```
32768 = 1000 0000 0000 0000b
```

`eax` ist ein 32-Bit-Register, also ist das Vorzeichenbit `0`. Beim arithmetischen Rechts-Shift wird das Vorzeichenbit dupliziert.

Das Ergebnis ist somit:

```
100 0000 0000b = 1024
```

Teil 2:

```
4711h =
0000 0000 0000 0000 0100 0111 0001 0001
1000 0000 0000 0000 0010 0011 1000 1000 ; Rechtsrot. um 1, CF=1
1111 1100 0000 0000 0000 0001 0001 1100 ; Ar. Re.-Sh. um 5, CF=0
1100 0000 0000 0000 0001 0001 1100 0111 ; Li-Rot. um 4 d. CF, CF=1
0000 1100 0000 0000 0000 0001 0001 1100 ; Re-Sh. um 4, CF=0
= C00011Ch
```

Lösung zu Kontrollaufgabe 2 auf Seite 22

Wir rechnen mit den 8-Bit-Registern `al` und `bl` beim Vergleichsbefehl

```
cmp al,bl ; also __ = al - bl
```

1. Beispiel `al = -112, bl = 64`
   $-112-64=-112+(-64)=10010000_2+11000000_2=(1)01010000_2$
   Damit gilt: $CF=1, SF=0, OF=1$, also $SF <> OF$
2. Beispiel `al = 112, bl = 64`
   $112-64=112+(-64)=01110000_2+11000000_2=(1)00110000_2$
   Damit gilt: $CF=1, SF=0, OF=0$, also $SF=OF$
3. Beispiel `al = 64, bl = -112`
   $64-(-112)=64+112=01000000_2+01110000_2=(0)10110000_2$
   Damit gilt: $CF=0, SF=1, OF=1$, also $SF=OF$
4. Beispiel `al = -64, bl = -112`
   $-64-(-112)=-64+112=11000000_2+01110000_2=(1)00110000_2$
   Damit gilt: $CF=1, SF=0, OF=0$, also $SF=OF$

wobei gilt:[14]

- Das Carryflag nach der Subtraktion ist in Klammern angegeben.
- Das Signflag entspricht dem höchstwertigen Bit des Ergebnisses.
- Das Overflowflag berechnet sich aus `xor` von Carryflag und dem Übertrag ins Vorzeichenbit.

[14] Siehe Abs. 3.

Lösung zu Kontrollaufgabe 3 auf Seite 29

Abb. 24: Kontrollaufgabe 3

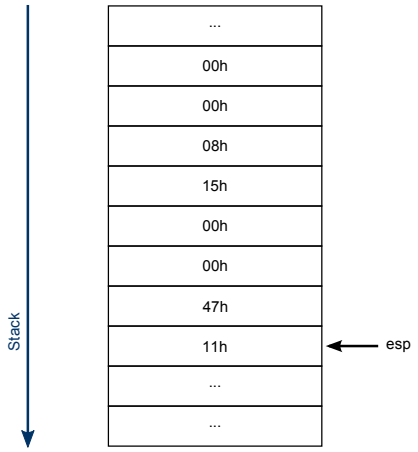


Abb. 24 zeigt die Situation auf dem Stack. Der Stack wächst von höheren Speicheradressen hin zu niedrigeren, im Bild von oben nach unten. Zuerst wird der Inhalt von `ebx` auf den Stack gelegt, also `0815h`, dann `4711h`. Wegen Little Endianness liegt jeweils das niederwertigste Byte an der kleinsten Speicheradresse. Beachten Sie, dass `eax` und `ebx` 32-Bit-Register sind, also vier Byte beinhalten.

Lösung zu Kontrollaufgabe 4 auf Seite 29

Durch `pop esp` würde der Stack Pointer mit `4711h` geladen. Der Stack käme in Unordnung und das anschließende `pop ebx` wäre ein undefinierter Speicherzugriff. `pop esp` ist ein unerlaubter Befehl.

Lösung zu Kontrollaufgabe 5 auf Seite 33

Abb. 25: Kontrollaufgabe 5

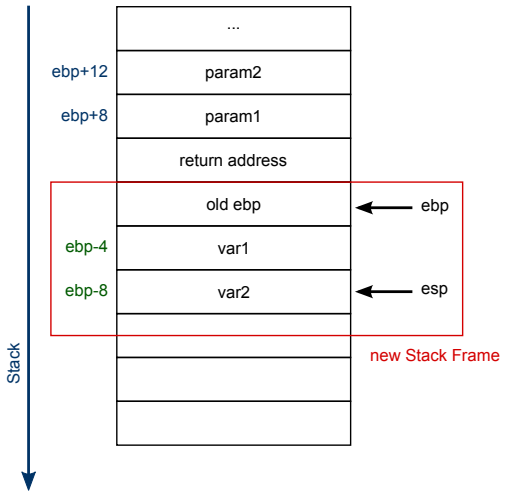


Abb. 25 zeigt die Situation auf dem Stack.

Lösung zu Kontrollaufgabe 6 auf Seite 33

```
push eax
push ecx
push ebp
mov ebp, esp
sub esp, 8            ; Reservierung von 8 Byte fuer 2 Integer
mov [ebp-4], 0        ; [ebp-4] referenziert var1
mov [ebp-8], 1        ; [ebp-8] referenziert var2
mov eax, [ebp-4]      ;
mov [ebp+16], eax     ; [ebp+16] referenziert param1
mov ecx, [ebp-8]      ;
mov [ebp+20], ecx     ; [ebp+20] referenziert param2
mov esp, ebp
pop ebp
pop ecx
pop eax
ret
```

`eax` und `ecx` werden zusätzlich auf dem Stack abgelegt. Dadurch ändert sich die Referenzierung von `param1` und `param2`. Bei der Referenzierung der lokalen Variablen ändert sich nichts.

Lösung zu Kontrollaufgabe 7 auf Seite 41

`5430h` wird in das Codesegment geschrieben. Falls der Code an Speicheradresse `1234` danach zur Ausführung kommt, handelt es sich um ein selbstmodifizierendes Programm.

Lösung zu Kontrollaufgabe 8 auf Seite 43

Bei einer Pagegröße von 4 KB können 2^{20} Pages angesprochen werden, bei einer Pagegröße von 4 MB sind es $2^{10} = 1024$.

Lösung zu Kontrollaufgabe 9 auf Seite 45

Die Präfixe `repe` und `repne` machen in Kombination mit den String-Vergleichsbefehlen `cmpsd` und `scasd` Sinn, da diese die Status-Flags setzen. Es können damit bspw. komplette Strings auf Gleichheit oder Ungleichheit geprüft werden (`repe/repne cmpsd`), oder es kann das Vorhandensein eines bestimmten Wertes in einem String geprüft werden (`repe/repne scasd`).

Verzeichnisse

I. Abbildungen

Abb. 1: Intel 80386-CPU . . . 8
Abb. 2: Registersatz des 80386 . . . 9
Abb. 3: Registerauf- teilung beim 80386 . . . 9
Abb. 4: Flags der 80386-CPU (Intel [1987]) . . . 11
Abb. 5: Intel- und AT&T-Syntax . . . 15
Abb. 6: Shift- und Rotationsbefehle . . . 19
Abb. 7: Endianness . . . 24
Abb. 8: Offset-Adressierung im Überblick . . . 27
Abb. 9: Stack-Prinzip mit push und pop . . . 28
Abb. 10: Auf- und Abbau eines Stack Frame . . . 31
Abb. 11: Aufbau des Stack Frame nach dem Prolog . . . 32
Abb. 12: Implementierung der Aufrufkonventionen in Assembler . . . 35
Abb. 13: Privilegienringe . . . 36
Abb. 14: Logischer, linearer und physikalischer Adressraum . . . 37
Abb. 15: Physikalischer Speicher eines Intel PC (Duarte [2008]) . . . 37
Abb. 16: Segmentierung im Protected Mode (Intel [2012]) . . . 40
Abb. 17: MMU: Pages und Frames (Freiling et al. [2010]) . . . 41
Abb. 18: Paging (80386) bei 4 KB großen Pages (Intel [2012]) . . . 42
Abb. 19: Adress- Umwandlung mit Segmentierung UND Paging (Intel [2012]) . . . 43
Abb. 20: Aufbau der Interrupt Vector Table . . . 47
Abb. 21: IA-32-Befehlsformat (Intel [2012]) . . . 48
Abb. 22: ModRM und SIB (Freiling et al. [2010]) . . . 48
Abb. 23: Befehlsformat . . . 50
Abb. 24: Kontrollaufgabe 3 . . . 56
Abb. 25: Kontrollaufgabe 5 . . . 56

II. Exkurse

Exkurs 1: **System-Flags** . . . 12
Exkurs 2: **Intel-Syntax vs. AT&T-Syntax** . . . 15
Exkurs 3: **Zweierkomplementdarstellung** . . . 21
Exkurs 4: **Befehlssynonyme** . . . 22

III. Tabellen

Tabelle 1: General Purpose Register . . . 10
Tabelle 2: Special Purpose Register . . . 10
Tabelle 3: Datentransferbefehle . . . 16
Tabelle 4: Arithmetische und logische Befehle . . . 17
Tabelle 5: Shift und Rotationsbefehle . . . 18
Tabelle 6: Bedingte Sprünge für vorzeichenlose Zahlen . . . 21
Tabelle 7: Bedingte Sprünge für vorzeichenbehaftete Zahlen . . . 22
Tabelle 8: Bedingte Sprünge bezogen auf einzelne Flags . . . 22
Tabelle 9: Adressie- rungsarten von IA-32 . . . 24
Tabelle 10: Subroutinen-Befehle . . . 28
Tabelle 11: Aufrufkonventionen . . . 34
Tabelle 12: StringBefehle und Präfixe . . . 45
Tabelle 13: Diverse Befehle . . . 46

IV. Literatur

Volker Claus and Andreas Schwill. *Duden Informatik*. Bibliographisches Institut, 2003.

Gustavo Duarte. *Motherboard Chipsets and the Memory Map*.
http://duartes.org/gustavo/blog/post/motherboard-chipsets-memory-map, 2008.

Felix Freiling, Ralf Hund, and Carsten Willems. *Software Reverse Engineering*. Vorlesung, Universität Mannheim, 2010.

Intel. *Intel 80386, Programmers Reference Manual*.
http://css.csail.mit.edu/6.858/2011/readings/i386.pdf, 1987.

Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*.
http://www.intel.com/content/www/us/en/architecture-and-technology/
64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html, 2012.

Kip R. Irvine. *Assembly Language for Intel-Based Computers (5th Edition)*. Prentice Hall, 2006.

Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C*. Hanser Fachbuch, 1990.

Joachim Rohde. *Assembler GE-PACKT, 2. Auflage*. Redline GmbH, Heidelberg, 2007.

Axel Stutz and Peter Klingebiel. *Übersicht über die C Standard-Bibliothek*.
http://www2.hs-fulda.de/~klingebiel/c-stdlib/index.htm, 1999.

Ray Wisman. *Processor Architecture*.
http://homepages.ius.edu/RWISMAN/C335/HTML/Chapter2.htm, 2012.