Die Assemblersprache der intel 80x86-Prozessoren

Prof. Dr. Klaus Wüst

Fachhochschule Gießen-Friedberg
Fachbereich MNI
Studiengang Informatik

Gießen im März 2003

Inhaltsverzeichnis

Vorwort

Das vorliegende Skriptum ist als Begleittext zur Vorlesung *Maschinennahe Programmierung* (später *Systemprogrammierung I*) an der Fachhochschule Gießen-Friedberg entstanden. Es soll die Hörer zumindest teilweise vom zeitraubenden Mitschreiben befreien und so die Vorstellung von mehr praktischen Beispielen ermöglichen. Der Hauptteil des Skriptums behandelt die Assemblersprache der Intel-Prozessoren der 80x86-Reihe. Die Hardware dieser Prozessoren kann hier nur wird nur soweit besprochen, wie es zum Verständnis der Programmierung erforderlich ist. Das Skriptum soll und kann nur exemplarisch sein, so sind z.B. nicht alle Befehle besprochen. Benutzen Sie daher ergänzend auch die einschlägige Literatur, einige Titel sind am Ende aufgeführt.

Für jede Art von Resonanz bin ich dankbar, das gilt ebenso für Verbesserungsvorschläge und Fehlerhinweise wie für positive Anmerkungen!

Gießen im März 2003 *Klaus Wüst*

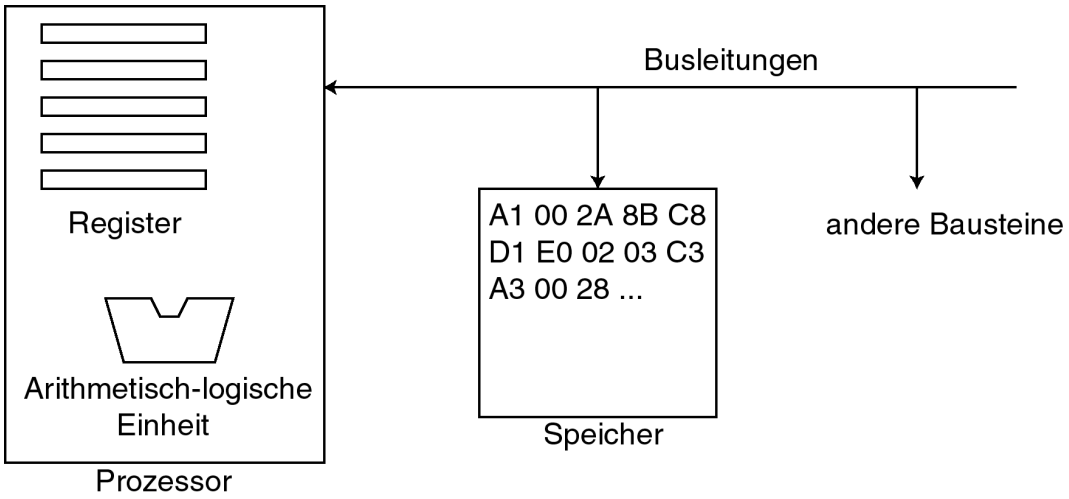
`Klaus.Wuest@mni.fh-giessen.de`

Kapitel 1

Einführung

1.1 Maschinencode und Assemblersprache

Das Herz eines jeden Computers ist der *Mikroprozessor* . Der Mikroprozessor kann Daten bearbeiten, d.h. verändern, sowie über ein Leitungssystem (Bus) mit Speicher- und Peripheriebausteinen austauschen. Für die Verarbeitung der Daten verfügt er über einige interne Speicherplätze, die sog. *Register* . Register sind Gruppen von Flipflops mit gemeinsamer Steuerung. Jedes Programm, das auf einem Computer abläuft, wird in viele kleine Einzelschritte zerlegt, die der Prozessor dann ausführt. Wie sehen diese Einzelschritte aus? Jeder Prozessor verfügt über einen gewissen Vorrat an Aktionen, den *Befehlssatz* Die Befehle des Befehlssatzes heißen auch die *Maschinenbefehle* . Es gibt Maschinenbefehle für den Datenaustausch mit Speicherzellen, für das Ansprechen von Peripheriegeräten, für den Transport zwischen Registern, für die bitweise Veränderung von Daten, für arithmetische Operationen an Daten und für vieles andere mehr.



Ein Mikroprozessor kann immer nur durch Maschinenbefehle angesteuert werden, alle anderen Bitmuster erkennt er nicht und verweigert die Arbeit. Ein Programm, das auf diesem Prozessor laufen soll, muss also so in Teilschritte zerlegt werden, dass sich jeder Teilschritt durch einen entsprechenden Maschinenbefehl umsetzen läßt.

Dazu ein Beispiel: Auf einem Mikroprozessor soll ausgeführt werden:

A =5*B + 1

Dies könnte z.B. wie folgt realisiert werden.

- Hole Inhalt der Speicherzelle B in Arbeitsregister 1
- Kopiere Inhalt des Arbeitsregisters 1 in Arbeitsregister 2
- Verschiebe Inhalt des Arbeitsregister 1 um zwei Bit nach links (entspricht der Multiplikation mit 4. Alternativ kann ein Multiplikationsbefehl benutzt werden, soweit vorhanden. Überlauf ist hier unberücksichtigt.)
- Addiere Inhalt von Arbeitsregister 2 zu Arbeitsregister 1 (entspricht jetzt 5*B)
- Inkrementiere Arbeitsregister 1
- Speichere Inhalt von Arbeitsregister 1 in Speicherzelle A

Für jede dieser Aktionen muss ein Maschinenbefehl zur Verfügung stehen. Wenn dann alle Aktionen als Maschinenbefehle formuliert sind, nennt man dieses Programmstück *Maschinencode* . Wie sieht nun Maschinencode aus? Maschinenbefehle sind einfach binäre Bitmuster in Einheiten zu 8 Bit, d.h. Bytes. Maschinencode ist also eine lange Folge von Einsen und Nullen, z.B.:

10100001 00000000 00101010 10001011 11011000 usw.

Die binäre Schreibweise nimmt zu viel Platz weg, man schreibt solche binären Daten fast immer hexadezimal auf. Die hexadezimale Schreibweise passt hier sehr gut, denn eine Hexadeziamlziffer stellt gerade 4 Bit dar, zwei Hexadezimalziffern also ein Byte. Unser Maschinencode sieht dann so aus:

A1 00 2A 8B D8 C1 E0 02 03 C3 40 A3 00 28

Diese Maschinenbefehle stehen dann im ausführbaren Programm, z.B. als .EXE-Datei. Zur Ausführung werden sie in den Speicher gebracht (geladen) und der Prozessor holt sich die Maschinenbefehle nacheinander aus dem Speicher. Jedes Byte wird dabei auf seine Bedeutung hin analysiert (dekodiert) und wenn ein gültiger Maschinenbefehl erkannt wurde, wird er ausgeführt. Wenn man das rein sequentielle Lesen unterbricht und stattdessen an einer anderen Stelle mit dem Einlesen fortfährt, wird das *Sprung* genannt. Durch Sprünge kann man Wiederholungen und Verzweigungen, die Grundelemente jeder Programmierung, realisieren. Zum Befehlssatz jedes Prozessors gehören daher auch Sprungbefehle. In den Maschinencode sind auch Operanden, d.h. Daten die direkt zum Befehl gehören, eingefügt. Theoretisch könnte man also mit Maschinencode Programme entwickeln, aber das macht man nur in Notfällen. Maschinencode hat doch einige schwere Nachteile:

- Die Programme sind sehr schlecht lesbar, man kann die Befehle nicht erkennen und keine Namen für Variablen und Sprungmarken vergeben.
- Die Programme sind sehr unflexibel, nach dem Einfügen von zusätzlichen Befehlen müßte man alle Sprungbefehle anpassen.
- Es können keine Kommentare eingefügt werden.

Diese Nachteile werden behoben durch die Einführung der *Assemblersprache* . In der Assemblersprache wird jeder Maschinenbefehl durch eine einprägsame Abkürzung mit typ. 3 Buchstaben dargestellt, das sog. *Mnemonic* . Die Assemblersprache wird dadurch relativ leicht lesbar und verständlich, stellt aber trotzdem ein vollständiges Abbild des Prozessors dar: Für jede Operation, die der Prozessor durchführen kann, gibt es einen zugehörigen Assemblerbefehl. Beispiele für Mnemonics, d.h. Assemblerbefehle, sind ADD für Addition, SHL für Shift left, MOV für Move. Operanden wie Registernamen, Konstante oder Variablen werden im Klartext genannt. Speicherplätze können frei wählbare Namen erhalten und damit wie Variablen in Hochsprachen benutzt werden. Ebenso können Sprungmarken Namen erhalten.

Wir wollen nun die oben stehende Liste von Aktionen zur Ausführung von A=5*B+1 in der Assemblersprache des intel 8086 aufschreiben. Die Speicherplätze heißen nun wirklich einfach A und B, als Register wurde AX und BX ausgewählt. Die benutzten Assemblerbefehle sind

mov bewegen, transportieren, Ziel zuerst genannt

shl shift left, Bitmuster nach links verschieben

add addieren, Summe kommt in ersten Operanden

inc inkrementieren, Wert um eins erhöhen

Das Assemblerprogramm(-stück) sieht dann so aus:

```
mov ax,B
mov bx,ax
shl ax,2
add ax,bx
inc ax
mov A,ax
```

Der *Assembler* (engl. Montierer) übersetzt dann das in Assemblersprache geschriebene Quellprogramm und erzeugt so den Maschinencode. In der folgenden Liste ist auf der rechten Seite aus den Assemblerbefehlen resultierende Maschinencode eingetragen Man sieht jetzt, wie der oben als Beispiel gegebene Maschinencode entstanden ist!

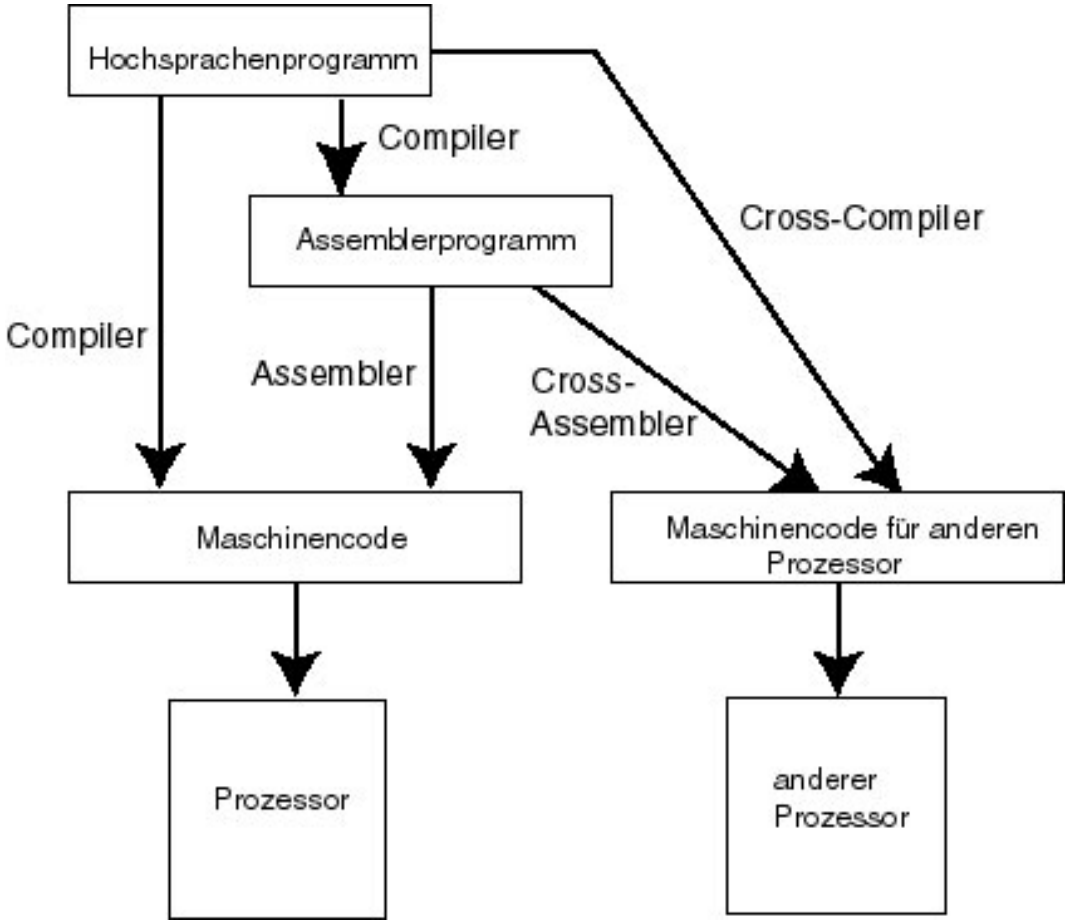
```
Assemblerbefehle        Daraus erzeugter Maschinencode

    mov ax,B            A1 002A
    mov bx,ax           8B D8
    shl ax,2            C1 E0 02
    add ax,bx           03 C3
    inc ax              40
    mov A,ax            A3 0028
```

Der Assembler-Programmierer muss sich nun nicht mehr um den Maschinencode kümmern und nur noch selten mit absoluten Adressen arbeiten. Trotzdem ist Assemblersprache eine direkte

Abbildung der Prozessorstruktur und die einzige Möglichkeit alle Fähigkeiten eines Prozessors zu nutzen. Jeder Assemblerbefehl erzeugt, im Gegensatz zu Hochsprachen, auch nur einen Maschinenbefehl.

Compiler erzeugen in der Regel direkt Maschinencode, manche Compiler können aber optional auch Assemblercode erzeugen. Maschinencode für andere Prozessoren erzeugt ein Cross-Assembler bzw, Cross-Compiler.



Wo liegen nun die Vor- und Nachteile von Assembler? Vorteile sind:

- Optimale Prozessorausnutzung möglich, guter Assemblercode ist sehr performant
- Vollständige Kontrolle über die Prozessorhardware
- Kompakter Code

Nachteile sind

- Der Programmierer braucht eine gute Kenntnis des Prozessors
- Jeder Prozessor hat seine eigene Assemblersprache, Spezialwissen erforderlich
- Reduzierte Portabilität

- Keine Bibliotheksfunktionen für Textausgabe, Dateioperationen, mathematische Funktionen, mathematische Ausdrücke u.ä.
- Fehler passieren etwas leichter und haben manchmal schwerwiegendere Folgen
- große Assemblerprogramme werden unhandlich

In der Praxis werden heute nur noch wenig Programme zu 100% in Assembler geschrieben. Meist schreibt man Programme in Hochsprachen und codiert sehr zeitkritische und sehr hardwarenahe Abschnitte in Assembler.

1.2 Register und Flags des 80386

1.2.1 Mikroprozessoren

In Abb.1.2.1 ist – stark vereinfacht – ein Mikroprozessorsystem dargestellt. Über die Adressleitungen wird im Hauptspeicher (und in anderen Bausteinen) die richtige Speicherzelle ausgewählt und über die Datenleitungen werden die Bitmuster vom und zum Prozessor transportiert. Die Steuerleitungen dienen dazu, von den parallel geschalteten Bausteinen immer den richtigen zu aktivieren.

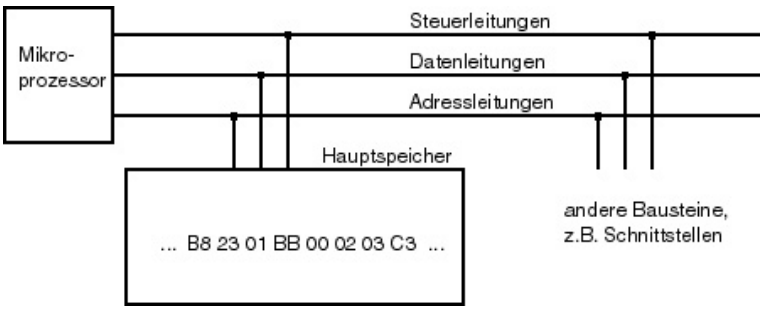


Abbildung 1.1: Grundsätzlicher Aufbau eines Mikroprozessorsystems

Die Bestandteile eines Prozessors lassen sich in vier Gruppen einteilen:

- Das Steuerwerk erzeugt die notwendigen Signale für die internen und externen Steuerleitungen (Busschnittstelle)
- Das Adresswerk erzeugt auf den Adressleitungen das notwendige Bitmuster, um die im Assemblerbefehl beschriebene Speicherzelle anzusprechen.
- Das Operationswerk führt die bitweisen und arithmetischen Operationen auf Datenoperanden aus
- Der Registersatz als spezieller Teil des Operationswerkes enthält eine gewisse Anzahl prozessorinterner Speicherzellen und Flags

1.2.2 Bits und Bytes

Ein Register ist eine Gruppe von Flipflops (1 Bit-Speicher) mit gemeinsamer Steuerung. Register umfassen meist 8, 16 oder 32 Bit. Eine Einheit aus 4 Bit heißt *Tetrade* oder *Nibble* , eine 8 Bit-Einheit heißt *Byte* . Ein *Wort* ist eine Dateneinheit, die die gleiche Größe hat wie das Hauptrechenregister des Prozessors. In der Welt der intel x86-Prozesoren wird häufig mit einem Wort eine 16 Bit-Einheit gemeint, weil der intel 8086 16 Bit-Allzweckregister hat. In diesem Zusammenhang ist ein Doppelwort dann eine 32-Bit-Einheit. Wichtig ist, dass eine Hexadezimale Ziffer gerade 4 Bit darstellt, ein Byte also genau durch zwei Hexziffern dargestellt wird usw.

Innerhalb einer Einheit sind die Bits nummeriert. Das niederwertigste Bit, das *Least significant Bit* , abgekürzt das *LSB* , ist immer Bit 0. Das höchstwertige Bit, das *Most significant Bit* , abgekürzt das *MSB* , ist bei einem Byte Bit 7, bei einem 16 Bit-Wort Bit 15 und bei einem 32 Bit-Wort Bit 31.

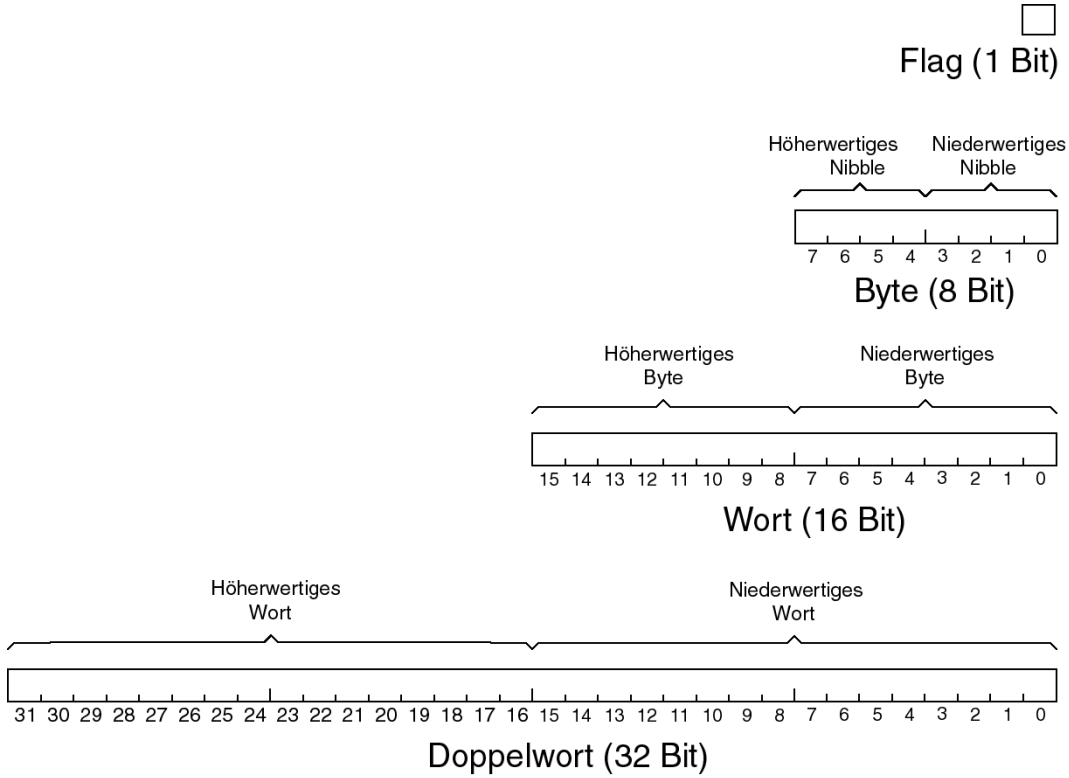


Abbildung 1.2: Die Dateneinheiten des intel 80386: Flags, Bytes, Worte und Doppelworte

1.2.3 Die Reihe der intel 80x86-Prozessoren

Im Jahre 1985 brachte intel den 8086-Prozessor auf den Markt, den ersten 16-Bit-Prozessor. Er erfuhr sehr starke Verbreitung, weil er im IBM-PC eingesetzt wurde, dem bald meistverbreiteten Mikrocomputer. Bei der Einführung des i8086 versuchte man, die Übertragung von 8-Bit-Programmen auf den neuen 16-Bit-Prozessor zu erleichtern und ermöglichte wahlweise den Zugriff auf die neuen 16-Bit-Register in zwei 8-Bit-Gruppen. So kann man das Hauptrechenregister AX wahlweise als zwei unabhängige 8-Bit-Register ansprechen: AL (Low Byte) und AH (High Byte). Der Befehl MOV AX,1234h ist absolut gleichwertig den beiden Befehlen MOV AH,12h + MOV AL,34h.

Der i8086 war der erste Prozessor einer langen und erfolgreichen Reihe, die Nachfolgetypen waren der i80186, i80286, i80386, i80486 und Pentium in vielen Varianten. Die Firma Intel hielt sich dabei streng an das Prinzip der *Aufwärtskompatibilität* , das bedeutet jeder neue Prozessor hat alle Funktionalität seiner Vorgänger und zusätzlich neue Features. So enthält z.B. ein Pentium-Prozessor in seinem Befehlssatz noch alle Befehle, die der 8086 hatte. Ebenso sind die ursprünglichen 16-Bit-Register weiterhin als Teilgruppe der jetzigen 32-Bit-Register (ab 386) verfügbar. Sogar das wahlweise Ansprechen der unteren 16 Bit in zwei 8-Bit-Gruppen ist immer noch möglich. Somit können ältere Programme ohne Veränderung des Maschinencodes unmittelbar auch auf den neueren Prozessoren laufen. Dieses Prinzip war für die Verbreitung der PC's und ihrer Software sehr wichtig.

Wir wollen hier vom intel 80386 (i386) ausgehen, der aus der Sicht eines Anwendungsprogrammierers schon nahezu die gleichen Register und Flags bietet, wie die Pentium-Prozessoren. In

Abb.1.2.3 sind die Register des i80386 gezeigt. Acht Registernamen beginnen mit einem E für extended, weil diese Register von 16 auf 32 Bit erweitert wurden. Für diese acht Register gilt, dass jeweils die unteren 16 Bit unter dem Namen des früheren 16-Bit-Registers separat angesprochen werden können. Also ist DI identisch mit den unteren 16 Bit von EDI, ebenso SI von ESI, SP von ESP und BP von EBP. Bei den vier Allzweckregistern EAX, EBX, ECX und EDX lassen sich die unteren 16 Bit als AX, BX, CX und DX ansprechen, und diese zusätzlich auch byteweise als AL und AH, BL und BH, CL und CH, DL und DH.

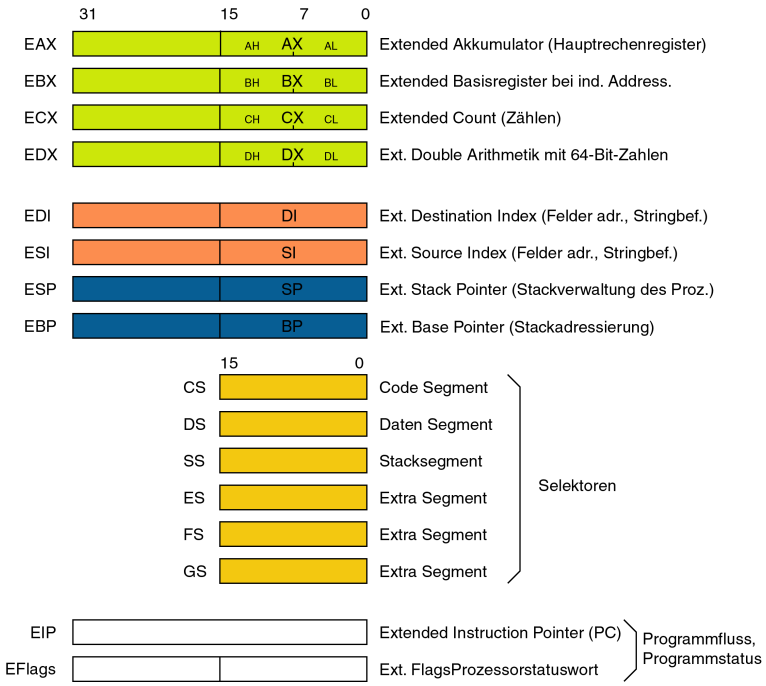


Abbildung 1.3: Die Register des intel 80386. Es sind nur die für die Anwendungsprogrammierung interessanten Register dargestellt.

Die Allzweckregister können relativ frei benutzt werden, bei einigen Befehlen werden allerdings bestimmte Register bevorzugt. So ist EAX das Hauptrechenregister (A=Accu), das bei einigen Rechenbefehlen zwingend benutzt werden muss und bei anderen günstig ist. ECX ist das Zählregister und wird bei Schleifen und Stringbefehlen zwingend als Zähler eingesetzt (C=Count). EDX wird mit EAX zusammen benutzt, um in EAX-EDX 64 Bit Operanden aufzunehmen, man hat dann also doppelte Bitzahl (D=Double).

EDI und ESI sind Register, die bei den sog. Stringbefehlen eine besondere Bedeutung als Zeigerregister haben. ESI (Extended Source Index) ist der Zeiger auf den Speicherplatz, der als Datenquelle dient, EDI (Extended Destination index) ist das Ziel.

EBP und ESP dienen zur Adressierung des *Stack* , eines besonderen Speicherbereiches, der als Last-in-First-out-Speicher organisiert ist. In ESP (Extended Stack Pointer) ist der Zeiger auf die aktuelle Spitze des Stack gespeichert, d.h. das zuletzt auf den Stack gebrachte Wort. Mit EBP wird der Stack frei adressiert.

Die Register CS, DS, SS, ES, FS und GS sind sogenannte Segmentregister. Beim 8086 war der Speicher nämlich segmentiert und CS enthielt einen Zeiger auf das Codesegment (Programmspeicher), DS einen Zeiger auf das Datensegment (Datenspeicher) und SS einen Zeiger auf das

Stacksegment (Stackspeicher). Im sog. *Protected Mode* (Verwaltung von geschützten Datenbereichen im Multitasking-Betrieb) dienen sie zur Aufnahme von Selektoren.

Das Register EIP (Extended Instruction Pointer) speichert die Adresse des nächsten auszuführenden Befehls im Programmcode. Ein Sprungbefehl hat zur Folge, dass einfach EIP neu geladen wird.

1.2.4 Flags

Das EFlag-Register unterscheidet sich völlig von den anderen Registern. Die Flipflops in diesen Registern werden nämlich einzeln gesteuert und jedes Flipflop hat eine ganz bestimmte Bedeutung, es ist ein *Flag* (Flagge, Fähnchen). Bei den Flags hat sich folgende Sprechweise eingebürgert:

- „Flag gesetzt“ bedeutet Flag=1; auch „ein Flag setzen“ (engl. to set the flag”)
- „Flag gelöscht“ bedeutet Flag=0; auch: „der Befehl löscht das Flag“ (engl. to clear the flag”)

Es gibt zwei Gruppen von Flags: Statusflags und Steuerflags.

Statusflags

Statusflags sind Flags, die der Prozessor nach arithmetischen oder bitweise logischen Operationen setzt, um etwas über das Resultat dieser Operation auszusagen. Der Programmierer kann diese Flags dann in bedingten Sprungbefehlen abfragen und Programmverzweigungen vom Zustand der Flags abhängig machen.

Zeroflag

Das Zeroflag, ZF, deutsch Nullflag, wird gesetzt, wenn das Ergebnis der letzten arithmetischen oder bitweise logischen Operation Null war. Beispiel:

```
mov ax,1            ;Zahlenwert 1 nach ax transportieren
dec ax              ;ax um 1 erniedrigen, Ergebnis ist Null
                    ;Zeroflag wird gesetzt
```

Signflag

Das Signflag, SF, Vorzeichenflag ist gesetzt, wenn das Ergebnis der letzten Operation negativ war. Beispiel:

```
mov ax,5            ;Zahlenwert 5 nach ax einschreiben
sub ax,7            ;7 von ax subtrahieren Ergebnis ist negativ
                    ;Signflag wird gesetzt
```

Carryflag

Das Carryflag, CF, Übertragsflag ist gesetzt, wenn bei der letzten Operation der *vorzeichenlose* Wertebereich überschritten wird. Anders ausgedrückt: wenn die Anzahl der vorhandenen Bits für das Ergebnis nicht ausreicht (s.auch Overflowflag) Beispiel:

```
mov al,250          ;Zahlenwert 250 nach al einschreiben
add al,10           ;10 zu al addieren. Ergebnis (260) überschreitet
                    ;den Wertebereich, da AL ein 8-Bit-Register ist und die
                    ;Werte 0..255 darstellen kann; Carryflag wird gesetzt
```

Overflowflag

Das Overflowflag, OF, Überlaufsflag ist gesetzt, wenn bei der letzten Operation der *vorzeichenbehaftete* Wertebereich überschritten wird. Im Gegensatz zum Carryflag betrifft das Overflowflag das Rechnen mit vorzeichenbehafteten Zahlen, also Zahlen die positiv und negativ sein können. Beispiel:

```
mov al,120          ;Zahlenwert 120 nach al einschreiben
add al,10           ;10 zu al addieren. Ergebnis (130) überschreitet
                    ;den Wertebereich, da AL ein 8-Bit-Register ist und die
                    ;Werte -128..+127 darstellen kann; Overflowflag wird gesetzt
```

Parityflag

Das Parityflag, PF, Paritätsflag wird gesetzt, wenn bei der letzten Operation ein Bitmuster entstanden ist, das in den *niederwertigen acht Bit* aus einer geraden Anzahl von Einsen besteht. Das Parityflag wird relativ selten benutzt, u.a. weil es nur acht Bit auswertet. Beispiel:

```
mov dl,110010b      ; binären Zahlenwert 110010 nach dl einschreiben
add dl,1            ;1 zu ax addieren. Das Ergebnis 110011 hat eine
                    ;gerade Anzahl von Einsen -> Parityflag wird gesetzt
```

Auxiliary Carry Flag

Das Auxiliary Carry Flag, AF, Hilfsübertragsflag, wird gesetzt, wenn bei der letzten Operation ein Übertrag von Bit 3 auf Bit 4, also ein Übertrag vom der unteren auf die obere Tetrade, entstanden ist. Dieses Flag ist nur beim Rechnen mit BCD-Zahlen nützlich und wird prozessorintern von den Ausrichtungsbefehlen benutzt.

Steuerflags

Steuerflags setzt das Programm bzw. der Programmierer, um die Arbeitsweise des Prozessors zu steuern.

Trap Flag

Das Trap Flag, TF, (manchmal auch Trace Flag) zu deutsch Fallenflag, wird hauptsächlich von Debuggern benutzt. Wenn das Trap Flag gesetzt ist, wird nach jedem ausgeführten Maschinenbefehl das Programm durch Interrupt Nr.4 unterbrochen. Die Interrupt-Behandlungsroutine gibt dann Informationen über Register, Flags und Speicher auf den Bildschirm, wie wir es bei einem Debugger gewohnt sind.

Interrupt Flag

Das Interrupt Flag, IF, Unterbrechungsflag, steuert, ob externe Unterbrechungen durch Hardwarebausteine zugelassen werden. In einem PC läuft die Bedienung der externen Geräte und Schnittstellen fast nur über Interrupts, weil dieses Konzept sehr effektiv ist. Das Interrupt Flag ist daher in der Regel gesetzt und wird nur in Ausnahmefällen für kurze Zeit gelöscht.

Direction Flag

Das Direction Flag, DF, Richtungsflag, wirkt nur auf eine ganz bestimmte Gruppe von Befehlen, die sog. *Stringbefehle* . Diese Stringbefehle verarbeiten gleich einen ganzen Block (oder String) von Daten. Dabei werden automatisch bestimmte Zeigerregister (EDI und ESI) benutzt und inkrementell verändert. Das Direction Flag steuert nun ob der Datenblock mit auf- oder absteigenden Adressen sequentiell bearbeitet wird.

1.3 Ein erstes Programm in Assembler

Der Aufbau eines Assemblerbefehles

Wir wollen nun ein erstes, einfaches Programm in Assembler schreiben, übersetzen und zum Ablauf bringen. Der Aufbau eines einzelnen Assemblerbefehles ist wie folgt:

`[Marke:] [Befehl] [Operande(en)] [;Kommentar]`

`[Marke:]` ist eine Bezeichnung für den Speicherplatz, an dem dieser Befehl später stehen wird; sie wird in der Regel für Sprungbefehle benutzt.

`[Befehl]` ist das Mnemonic des Assemblerbefehls, z.B. MOV oder INC. Die Mnemonics dürfen beliebig klein oder groß geschrieben werden.

`[Operand(en)]` sind ein, zwei oder drei Operanden, je nach Befehl. Es gibt auch Befehle, die keine Operanden brauchen. Auch Register- und Speicherplatzbezeichnungen dürfen beliebig klein oder groß geschrieben werden.

`[;Kommentar]` ist eine Kommentar zu dieser Zeile, er kann am Ende der Zeile nach einem Semikolon angefügt werden.

Alle Bestandteile sind in eckigen Klammern aufgeführt, weil sie unter Umständen entfallen können. Erlaubt sind auch reine Kommentarzeilen und Leerzeilen.

Alternativ zu den Befehlszeilen gibt es *Direktiven* . Diese werden nicht in ausführbaren Maschinencode umgesetzt, sondern steuern nur den Assembler beim Übersetzungslauf.

Unser erstes Programm

```
;
;  Programm HELLO.ASM
;
    .MODEL SMALL            ; Speichermodell "SMALL"; Die maximale Größe
                            ; von Datensegment und Codesegment ist je 64kB
    .STACK 100h             ; 256 Byte Stack reservieren

   .DATA                    ; Beginn des Datenbereichs
   Meldung      DB 'Hallo Welt',13,10,'$'   ; Reservierung von Speicherplatz
                            ; und Vorbelegung mit Text und Steuerzeichen
                            ; 13 = Carriage Return d.h. Rücksprung in Spalte 1
                            ; 10 = Line Feed d.h. Zeilenvorschub

    .CODE                    ; Beginn des Codebereichs (Assemblerbefehle)
Programmstart:               ; Label haben einen Doppelpunkt am Ende
   mov  ax,@data             ; Uebergabe der Adresse des Datensegments
                             ; zur Laufzeit
   mov  ds,ax                ; DS zeigt nun auf das Datensegment
```

```
   mov  ah,9                  ; DOS-Funktion, die einen durch '$' begrenzten
                              ; String auf den Bildschirm ausgibt
   mov  dx,OFFSET Meldung     ; Offset der Adresse des Strings
   int  21h                   ; Interrupt 21h : Aufruf von DOS

; Programmende, die Kontrolle muss explizit an DOS zurueckgegeben werden
   mov  ah,04Ch               ; ah=04C : DOS-Funktion "terminate the program"
   mov  al,0                  ; DOS-Return-Code 0
   int  21h                   ; Interrupt 21h : Aufruf von DOS
   END Programmstart          ; END = Ende der Übersetzung,
                              ; danach Angabe des Einsprungpunktes
```

Nun also zu unserem ersten kleinen Programm, es gibt traditionell die Worte „Hallo Welt“ auf den Bildschirm aus. [1] Es beginnt mit einer Kommentarzeile, die den Namen und eine kurze Beschreibung des Programms enthält. Danach folgt mit der Direktive „.MODEL“ eine Festlegung des Speichermodells, d.h. eine Angabe darüber, wieviel Platz für Programmcode und Daten maximal gebraucht wird. Speichermodelle werden nur in DOS-Umgebungen gebraucht. (s.Abschn. 2 Die Direktive „.STACK“ legt fest, wie viel Speicherplatz für den *Stack* reserviert wird, einen besonderen Bereich des Hauptspeichers, der als Zwischenspeicher in fast jedem Programm benutzt wird. Nach der Direktive „.DATA“ wird der Datenbereich angelegt. Dabei wird nicht nur Platz für Daten reserviert, sondern den Speicherplätzen werden auch Namen und optional Vorbelegungswerte zugeordnet. Im Beispielprogramm wird eine Variable mit dem Namen „Meldung“ angelegt und mit einer Kette von Buchstaben und Steuerzeichen vorbelegt. Nach „.CODE“ folgt der Code- d.h. Programmbereich mit den Assemblerbefehlen. Die beiden ersten Befehle dienen dazu, die (Segment-)Adresse des Datenbereiches zur Laufzeit ins DS-Register zu übertragen.

Da wir die Zeichen nicht einzeln in den Bildschirmspeicher schreiben wollen, nehmen wir danach einen Betriebssytemaufruf (Funktion 9 von Int 21h) zu Hilfe. Danach folgt schon ein Betriebssytemaufruf zur Beendigung des Programmes.

Es wurden nur zwei Assemblerbefehle benutzt: MOV zum Datentransport und INT für die Betriebssystemaufrufe.

Wenn das Programm fertig editiert ist, wird es als Assembler-Quelldatei gespeichert, d.h. als .ASM-Datei, z.B. `HALLO.ASM`. Danach wird sie assembliert, im Falle des Borland-Assmblers mit

`TASM Dateiname.ASM` oder einfach `TASM Dateiname`

in unserem Beispiel also `TLINK HALLO`. Der Assembler übersetzt und prüft nun das geschriebene Programm. Wenn es keine schweren Fehler mehr enthält, erzeugt er eine Zwischendatei, die sog. *Objektdatei* , in unserem Fall also `HALLO.OBJ`. Diese enthält schon den erzeugten Maschinencode aber noch nicht die richtigen Adressen für Unterprogramme und Sprungmarken. Um auch das Zusammenbinden mehrerer Assembler- und Hochsprachenprogramme zu ermöglichen, gibt es ein weiteres Werkzeug, den Binder, engl. *Linker* . Im nächsten Schritt wird also der Linker aufgerufen, um den Objektfile zu binden (linken):

`TLINK Dateiname.OBJ` oder einfach `TLINK Dateiname` in

[1] In dem gezeigten Listing werden die sogenannten vereinfachten Segmentdirektiven des Borland Assemblers benutzt. Eine andere Möglichkeit ist die direkte Definition der Segmente mit „SEGMENT“ und „ENDS“

In unserem Fall muss es also heißen `TLINK HALLO`. Danach hat der Linker eine ausführbare Datei erzeugt: `HALLO.EXE` kann durch Aufruf auf der Kommandozeile (`HALLO` Return) oder durch Anklicken auf einer Windows-Oberfläche gestartet werden.

Kapitel 2

Organisation und Benutzung des Hauptspeichers im Real-Mode

Der Real Mode ist die einfache Betriebsart, in der die Intel-Prozessoren so wie der Urvater 8086 arbeiten. Im Unterschied dazu wird im Protected Mode der Speicher völlig anders verwaltet. Auf den Protected Mode kann hier nicht eingegangen werden. In diesem Kapitel werden die Methoden des Speicherzugriffs behandelt, d.h. das Schreiben in den Speicher und das Lesen aus dem Speicher.

2.1 Speichervariablen definieren

Am Anfang des Programmes besteht die Möglichkeit Speichervariablen unter Angabe von Namen und Typ zu definieren. Das bewirkt:

1. Es wird dann in einem bestimmten Bereich des Speichers, dem *Datensegment*, für jede Variable die dem Typ entsprechende Anzahl von Bytes freigehalten,
2. Dieser Bereich kann später unter dem Namen der Variablen angesprochen werden.

Der Typ der definierten Variable wird durch eine entsprechende Direktive, d.h. Anweisung an den Übersetzer, festgelegt:

Direktive	Name	Anzahl Byte einer Einheit	Beispiele für Verwendung
DB	Define Byte	1	8-Bit-Variable (char), Strings
DW	Define Word	2	16-Bit-Variable (integer), NEAR pointer
DD	Define Doubleword	4	32-Bit-Variable (long), Far pointer
DQ	Define Quadword	8	Gleitkommazahl
DT	Define Tenbyte	10	BCD-Zahlen

Bei der Definition kann gleichzeitig eine *Initialisierung* d.h. *Vorbelegung* vereinbart werden, d.h. daß der definierte Speicherplatz beim Programmstart einem gewünschten Wert vorbelegt wird. Der Initialisierungswert kann dezimal, hexadezimal, oktal, binär oder als Character angegeben werden. Bei Verzicht auf Initialisierung wird ein "?" eingetragen.

Die Syntax für die Definition einer Einzelvariablen ist also:

Variablenname Define-Direktive ?/Wert

Beispiele:

```
Zaehler1 DB ?   ;Def. der Byte-Variablen Zaehler1,keine Vorbesetzung
Zaehler2 DB 0   ;Def. der Byte-Variablen Zaehler2, Vorbesetzung mit 0
Endechar DB ?   ;Auch Zeichen werden als Byte definiert
Startchar DB 65 ;Vorbesetzung mit ASCII-Zeichen #65 = 'A'
Startchar DB 'A';gleiche Wirkung, besser lesbar
Regmaske DB 00110101b                ;Vorbesetzung mit binärem Wert (Bitmuster)
Pixelx DW ?     ;Wort-Variable ohne Vorbesetzung
Pixely DW 01AFh ;Wort-Variable, Vorbesetzung mit hexadezimalem Wert
Schluessel DD ? ;Doppelwort-Speichervariable (32 Bit)
Quadrate1 DQ ?  ;Quadword-Variable
zehnbytes DT ?  ;Tenbyte-Variable
```

Mit einer Anweisung können auch gleich mehrere Speicherplätze gleichen Typs, also *Felder* , definiert werden. Das geht auf zwei Arten:

1. Durch die Angabe eines Zahlenwertes und das Wort `DUP` (=Duplizieren), wobei die mit DUP angelegten Felder einheitlich initialisiert werden können.
2. durch Aufzählungen bei der Vorbesetzung, wobei die Anzahl der aufgezählten Elemente gleichzeitig die Feldgröße festlegt. Dies ist speziell bei Texten nützlich.

Die Syntax für die Definition einer Feldvariablen mit DUP ist:

Variablenname Define-Direktive Anzahl Feldelemente DUP (Vorbesetzungswert)/(?)

Beispiele:

```
Meldung1 DB 80 DUP(?)               ;Feld aus 80 Bytes, keine Vorbelegung
Quadrate1 DD 100 DUP(0)             ;Feld aus 100 Doppelworten, Vorbelegung mit 0
```

Bei der Felddefinition durch Aufzählung bei der Vorbesetzung entfällt die Angabe DUP. Beipiele:

```
Meldung1 DB 'Divisionsfehler!'
               ;Vorbesetzung mit einer Zeichenkette,das Feld erhält 16 Byte Speicherplatz
Meldung1 DB 'Hallo Welt',13,10
               ;Vorbesetzung mit einer Zeichenkette, und Steuerzeichen, 12 Byte Speicherplatz
Quadrate2 DD 1,4,9,16,25,36
         DD 49,64,81,100
               ;initialisiertes Doppelwortfeld mit Zeilenumbruch
```

Wichtig: Bei Feldern repräsentiert der Name des Feldes die Adresse des ersten Speicherplatzes. Um darauffolgende Speicherplätze anzusprechen, benutzt man die indirekte Adressierung.

Die Daten liegen im Datensegment in der Reihenfolge ihrer Definition. Der Speicher ist – auch heute noch – in Bytes organisiert. Bei der Speicherung von Texten gibt es daher keine Probleme, denn die Textzeichen sind ja 8-Bit-Größen. Anders ist das bei Zahlen: Zahlen, die mit mehr als 8 Bit dargestellt sind, müssen mehrere Speicherzellen (Bytes) belegen. dabei zeigen die Intel-Prozessoren eine interessante und verwirrende Eigenart: Sie speichern Daten im sogenannten *Little Endian-Format* . Dabei gilt das Prinzip: *„Lowest Byte first“*; alle Darstellungen von Zahlen werden in Bytes zerlegt und diese Bytes werden beginnend mit dem niedrigstwertigen Byte im Speicher abgelegt. Die Zahl 120h in 16-Bit-Darstellung steht also im Register als `0120h` und im Speicher als `20h 01h`. Die 32-Bit-Zahl 01304263h liegt im Speicher als `63h 42h 30h 01h`. Zum Glück erledigt der Prozessor die Arbeit des Umdrehens: Er verdreht die Reihenfolge der Bytes beim Ablegen in den Speicher und bringt sie beim Zurückholen ins Register wieder in die richtige Ordnung. Als Assemblerprogrammierer muss man Little Endian nur kennen, falls man einmal einen Speicherdump auswertet oder auf die genauen Byte-Positionen Bezug nimmt.

Beispiel:

```
Bytevar DB 90h
Wordvar DW 4501h
Dwordvar DD 12345678h
Stringvar DB 'ABCDEFGH',13,10,'$'
```

Die Variablen werden wie folgt im Speicher abgelegt:

90	01	45	78	56	34	12	41	42	43	44	45	46	47	48	0D	0A	24

Ein wichtiger Begriff ist der sog. *Offset* . Dies bedeutet soviel wie Abstand vom Segmentanfang oder relative Adresse im Datensegment. So hat zum Beispiel die erste Variable, in unserem Beispiel `Bytevar`, den Offset 0. Da `Bytevar` nur ein Byte belegt, hat `Wordvar` den Offset 1. `Wordvar` belegt 2 Byte, also hat die nächst folgende Variable `Dwordvar` den Offset 3, Stringvar den Offset 7. In vielen Fällen, z.B. bei Aufruf der DOS-Funktion 9 (Stringausgabe) ist es notwendig, in einem Register den Offset einer Variablen zu übergeben. Man überlässt dem Assembler das Abzählen, indem man den Operator “Offset“ benutzt. Für obige Datendefinition z.B. :

```
mov ah,9                    ;DOS-Funktion Stringausgabe
mov dx, Offset Stringvar    ;besser als mov dx, offset 7
                            ;Offset der Stringvariablen nach DX
int 21h                     ; Systemaufruf und Ausgabe }
```

Einige Beispiele zur Benutzung des Hauptspeichers:

```
.DATA
Zaehler1 DB ?
Zaehler2 DB 0
Endechar DB ?
```

```
   Startchar DB 'A'
   Pixelx DW ?
   Pixely DW 01FFh
   Schluessel DD 1200h
.CODE
   mov Zaehler1, 0      ; Direktwert auf Speichervariable schreiben
   mov Zaehler2, al     ; 8-Bit-Registerwert auf Speichervariable kopieren

   mov ah,2
   mov dl, Startchar    ; 8-Bit-Speichervariable lesen und in Register kopieren
   int 21h

   mov Endechar,'Q'     ; Direktwert als Character angeben und auf Speichervar. schr.
   xchg cx, Pixely      ; 16-Bit-Speichervariable mit Registerinhalt austauschen

   mov schluessel,eax   ; 32-Bit-Register auf Speichervariable kopieren
   movzx edi,Pixelx     ; 16-Bit-Speichervariable in 32-Bit-Register kopieren,
                        ; höherwertiges Wort des Registers dabei auf Null setzen
```

Weitere Beispiele dazu in Abschn. 2.4.3!

2.2 16-Bit-Umgebungen: Der segmentierte Hauptspeicher

Der Intel8086 hat einen Adressbus aus 20 Adressleitungen. mit dem er $2^{20} = 1MB$ Hauptspeicher adressieren (ansprechen) kann. Das auf den Adressleitungen anliegende Bitmuster, die *physikalische Adresse* , umfasst also 20 Bit. Wie wir wissen, hat der Intel8086 aber keine Register mit mehr als 16 Bit. Er kann also eine physikalische Adresse nicht in einem Register speichern! Mit seinen 16-Bit-Registern kann er nur einen Speicher von $2^{16} = 64kB$ direkt adressieren. Man hat das Problem wie folgt gelöst:

- Bei den Prozessoren i8086 bis i80286 wird der Inhalt zweier 16-Bit-Register kombiniert, um eine 20-Bit-Adresse zu erhalten.
- Ab dem i80386 stehen 32-Bit-Register zur Verfügung.

Der zweite Punkt wird im nächsten Abschnitt behandelt.

Der physikalische Adressraum ist

$$0 \ldots 2^{20} - 1 = 0 \ldots FFFFh = 0 \ldots 1048575d$$

um diese physikalischen Adressen zu bilden, werden zwei 16-Bit-Anteile, *Segment* und *Offset* , zu einer 20-Bit-Adresse zusammengefügt nach der Formel:

$$\textit{Physikalische Adresse} = 16 * \textit{Segment} + \textit{Offset}$$

Diese Berechnung wird automatisch innerhalb des Prozessors im Adresswerk durchgeführt. Das Adresswerk arbeitet daher nach folgendem Schema:

16-Bit-Segment	0 0 0 0
0 0 0 0	16-Bit-Offset
20-Bit-Physikalische Adresse	

Der Segmentanteil dabei wird immer aus einem der Segmentregister genommen. Der Offset kann sehr flexibel gebildet werden: Man kann ihn zur Laufzeit berechnen lassen als Summe aus zwei Registern und einer Konstante. (s.Abschn. 2.4). Das Paar aus Segment und Offset heißt auch *logische Adresse* . Logische Adressen werden meist mit einem Doppelpunkt geschrieben. Beispiel: B800:0200 ist die Adresse mit dem Segmentanteil B800h und dem Offset 200h.

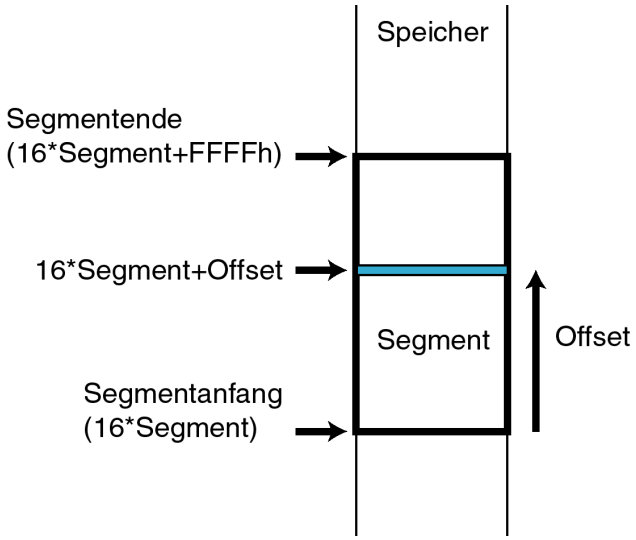


Abbildung 2.1: Aufbau eines 16-Bit-Segmentes

Die obige Berechnungsvorschrift hat folgende Konsequenzen:

1. Es gibt Adressen, die man allein durch Veränderung des Offset-Anteils bilden kann: Nahe oder „NEAR“-Adressen.
2. Für weiter entfernt liegende Adressen muss auch das Segmentregister umgesetzt werden: Weite oder „FAR“-Adressen.
3. Der Adressraum, den man ohne Änderung eines Segmentregisters erreichen kann, ist 64 kB groß und heißt *Segment* . Der Offset ist dabei die Position des Speicherplatzes relativ zum Segmentanfang. Eine andere Bezeichnung für Offset ist effektive Adresse.
4. Der Segmentanfang fällt immer auf ein Vielfaches von 16, 16 Byte sind ein *Paragraph* .
5. Zu einer logischen Adresse gibt es immer eine eindeutige physikalische Adresse, umgekehrt gilt das nicht. Beispiel:

Segment	B800h	Segment	B920h
Offset	1234h	Offset	0034h
Physikalische Adresse	B9234h	Physikalische Adresse	B9234h

6. Für das verwendete Segmentregister werden automatische Vorgaben benutzt:

   Bei Zugriff auf Programmcode: CS:IP
   Bei Zugriff auf Daten (i.d.R.): DS
   Alle Befehle die den Stack benutzen (CALL,RET,RETI,PUSH,POP, sowie Zugriffe mit Adressierung über BP): SS
   Zieloperand bei Stringbefehlen: ES

   Soll ein anderes Segmentregister benutzt werden, muss dies im Befehl ausdrücklich angegeben werden: *Segment Override*

```
mov ax,[bx+10]  ;Default-Segmentregister DS wird benutzt:
                ;Speicherzelle DS:BX+10 nach Register AX laden.

mov ax,ES:[bx+10]Speicherzelle ES:BX+10 nach Register AX laden.
```

Diese Speicherorganisation hat für die Programmierung natürlich Konsequenzen: Adressen, die innerhalb des Segments liegen (NEAR) sind durch Einsetzen des richtigen Offsets bequem zu erreichen. Wenn ich also in einem Programm einen Datenbereich von bis zu 64 kB habe, passt alles in ein Segment, alle Variablen können ohne Veränderung des Segmentregisters erreicht werden. Für die Angabe einer Adresse genügt dann der Offsetanteil. In Hochsprachen werden Adressen meist als *Zeiger* oder *Pointer* bezeichnet. Zeiger, die nur den Offset beinhalten heißen *NEAR-Zeiger* bzw. *NEAR-Pointer* .

Adressen ausserhalb des Segments machen mehr Umstände: Das Segmentregister muss vor dem Speicherzugriff umgesetzt werden. Verwaltet man also eine Datenmenge von mehr als 64 kB, so muss für jede Adresse Segment *und* Offset angegeben werden. Jedesmal wenn eine Adresse gespeichert oder geladen wird, müssen also zwei Anteile behandelt werden. Das macht die Programme nun umständlicher und langsamer. [1] Zeiger die Offset und Segment enthalten heißen *FAR-Zeiger* bzw. *FAR-Pointer* .

Die Unterscheidung zwischen NEAR- oder FAR-Adressen gibt es auch beim Programmcode. So muss z.B. bei einem Sprungbefehl in einem kleinen Programm (weniger als 64 kB) nur das IP-Register umgesetzt werden. In einem großen Programm muss zusätzlich das CS-Register umgesetzt werden. Das führte zur Entwicklung der sog. *Speichermodelle* . Ein Speichermodell legt vor der Übersetzung fest, wieviel Platz für Daten und Code zur Verfügung steht. In Segmenten bis 64 kB kann dann sowohl bei Code als auch bei Daten mit NEAR-Zeigern gearbeitet werden, in größeren mit FAR-Zeigern. Durch die Wahl des richtigen Speichermodells versucht man also, möglichst effiziente Programme zu erstellen. Der Turbo-Assembler von Borland unterstützt folgende Speichermodelle:

Name	Programmcode	Daten
TINY	zusammen 64 kB	
SMALL	bis zu 64 kB	bis zu 64 kB
MEDIUM	mehr als 64 kB	bis zu 64 kB
COMPACT	bis zu 64 kB	mehr als 64 kB
LARGE	mehr als 64 kB	mehr als 64 kB
HUGE	mehr als 64 kB	mehr als 64 kB

[1] Für das Laden eines Zeigers in einem Schritt gibt es zwei Spezialbefehle: LDS und LES. Für das Speichern von Zeigern gibt es keine Spezialbefehle.

Speichermodell HUGE unterscheidet sich von LARGE dadurch, daß es einzelne Datenbereiche mit einer Größe von mehr als 64 kB unterstützt. Die gleichen Speichermodelle findet man z.B. auch in Borland C–Programmen. Das Speichermodell wird am Anfang unserer Assemblerprogramme mit der Anweisung `.MODEL` eingestellt.

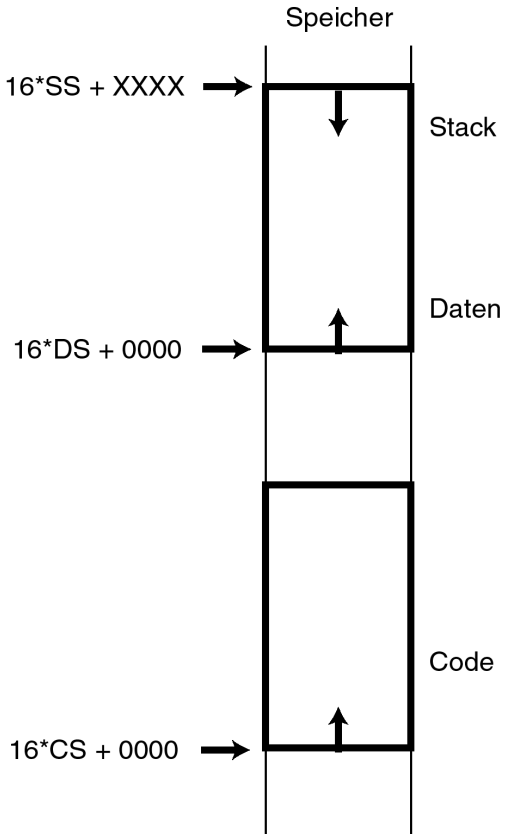


Abbildung 2.2: Angelegte 16-Bit-Segmente bei einem Programm im Speichermodell SMALL. Beide Segmente sind 64 kB groß

Um einen Speicherplatz absolut zu adressieren, muss man einen entsprechenden FAR-Zeiger aufsetzen. Nehmen wir z.B. an, dass der Speicherplatz mit der physikalischen Adresse B8000h mit einem 'A' beschrieben werden soll. Dann könnte man mit dem Segmentanteil B000h und dem Offset 8000h arbeiten. Genausogut könnte man Segmentanteil B800h und Offset 0 verwenden. Der Segmentanteil *muss* dann in ein Segmentregister geschrieben werden. Einfach ist hierbei die Verwendung von ES mit anschliessendem Segment Override. Der Programmcode könnte sein:

```
mov ax,B000h          ;ES mit Segmentanteil laden
mov es,ax
mov bx,8000h          ;Offset nach BX, FAR-Zeiger liegt in ES:BX
mov es:[bx],'A'       Zugriff mit FAR-Zeiger
```

Das DS-Register kann man ohne Segment Override benutzen. Man muss es aber retten und nachher wiederherstellen, damit es auf das angelegte Datensegment verweist:

```
mov ax,B000h          ;ES mit Segmentanteil laden
```

```
push ds              ; DS auf den Stack retten
mov ds,ax
mov bx,8000h         ;Offset nach BX, FAR-Zeiger liegt in DS:BX
mov [bx],'A'         ;Zugriff mit FAR-Zeiger
pop ds               ;DS-Register restaurieren
```

Die hier beschriebene Speicherorganisation bezeichnet man auch als *segmentierten Speicher* . Beim Arbeiten unter DOS hat man es immer mit dem segmentierten Speicher und mit FAR-Zeigern zu tun. Der segmentierte Speicher mit seinen NEAR- und FAR-Adressen sowie den unterschiedlichen Speichermodellen wurde bald als hinderlich empfunden.

2.3 32-Bit-Umgebungen: Der unsegmentierte Hauptspeicher

Mit der Einführung des i80386 standen 32-Bit-Register zur Verfügung. Mit einem 32-Bit-Zeiger kann man $2^{32} = 4GB$ Speicher adressieren. Dies übersteigt bislang immer den in einem PC tatsächlich installierten Hauptspeicher, also kann jede beliebige Hauptspeicheradresse in einem Register gespeichert werden. Segmente, NEAR- und FAR-Adressen sowie Speichermodelle gehören der Vergangenheit an, jede Adresse wird durch einen einzigen 32-Bit-Wert beschrieben, den 32-Bit-Offset. Anders ausgedrückt: der ganze Speicher stellt ein einziges großes Segment dar, man hat ein sog. *flaches Speichermodell* . [2] In einer 32-Bit-Umgebung können leicht Datenstrukturen verwaltet werden, die größer als 64 kB sind. Im folgenden Beispiel wird ein Feld von 100000 Byte mit FFh beschrieben.

```
                mov ebx,0            ;Zeiger initialisieren
l1:             mov [ebx],0FFFFFFFFh ;4 Byte in Feld schreiben
                add ebx,4            ;Zeiger weiterrücken
                cmp ebx,100000       ;Schleifenende?
                jbe l1
```

2.4 Adressierungsarten

2.4.1 Unmittelbare Adressierung

Bei der unmittelbaren Adressierung (immediate adressing) steht der Quelloperand unmittelbar im Befehl. Er wird bei Übersetzung fest in den Maschinencode eingebunden und folgt unmittelbar auf den Befehlscode. Beispiel:

```
mov bx,9
```

wird im Maschinencode zu `B0 09`.

[2] Auch der 386 und seine Nachfolger unterstützen weiterhin Segmente, diese sind aber mehr zur Verwaltung des Multitaskings gedacht und sind mit mehreren Schutzmechanismen ausgestattet.

2.4.2 Registeradressierung

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

```
mov ebx,ebp
```

2.4.3 Direkte Speicheradressierung

Bei der direkten Speicheradressierung wird der Offset des adressierten Speicherplatzes direkt angegeben und liegt nach der Assemblierung fest. Die Angabe des Offsets kann als konstante Zahl oder (besser) als Variablenname erfolgen. Der Variablenname kann – muss aber nicht – in eckigen Klammern eingeschlossen sein. Es können auch Feldelemente direkt adressiert werden, indem nach dem Feldnamen ein Pluszeichen und eine Konstante folgen. Beispiele:

```
mov ax, Zaehler1     ;Direkte Adressierung ohne eckige Klammern
mov [bigcount],ecx   ;Direkte Adressierung mit eckigen Klammern
mov ecx, [Feld+2]    ;Direkte Adr. von Feld + 2 Byte
mov al,[0020]        ;Direkte Adressierung über Offsetwert
                     ;schlecht und von manchen Assemblern beanstandet
```

Achtung: Die Intel80x86-Prozessoren können in jedem Befehl nur einen Speicheroperanden adressieren! Es geht also nicht: `mov [Variable1],[Variable2]`

2.4.4 Die indirekte Speicheradressierung

Die direkte Adressierung reicht nicht mehr aus, wenn die Adresse der Speicherzelle erst zur Laufzeit bestimmt wird. Das kommt z.B. bei Feldern häufig vor. Nehmen wir z.B. folgende Aufgabenstellung:

Für eine Zeichenkette soll die Häufigkeit der darin vorkommenden Zeichen bestimmt werden. Man braucht ein weiteres Feld um die Häufigkeit jedes Zeichens abzuspeichern. Bei der Bestimmung der Häufigkeit muss für jedes erkannte Zeichen der dazu gehörende Häufigkeitszähler um eins erhöht werden. Auf welchen Speicherplatz zugegriffen wird, ergibt sich also erst zur Laufzeit und hängt von den Daten ab. Eine direkte Adressierung, wie z.B. `inc [Haufigkeit+5]` usw. reicht nicht aus. Ebenso liegt der Fall bei der Programmierung von Sortieralgorithmen und vielen anderen Problemstellungen der Informatik.

Bei dem Problem der Häufigkeitsbestimmung wäre nach den Deklarationen

```
.DATA Zeichenkette DB 'ABCDEfGH' Haeufigkeit DB 26 DUP (0)
```

im Codesegment eine direkte Adressierung wie z.B.

```
inc [Haeufigkeit+3]
```

nicht zweckmäßig, sie würde immer das Feldelement Nr.3 (das vierte) ansprechen. Man müßte statt der 6 etwas Variables einsetzen können.

Genau dies erlaubt die *Register-indirekte Adressierung* , auch kurz *indirekte Adressierung* . Mit den Befehlen

```
mov bx, 3            ;Vorbereitung
inc [Haeufigkeit+bx] ;indirekte Adressierung
```

wird nun auch das Feldelement Nr.3 angesprochen, hier kann man aber zur Laufzeit berechnen, welcher Speicherplatz angesprochen wird!

Die indirekte Adressierung bietet die Möglichkeit, den Offset *zur Laufzeit flexibel zu berechnen*, und zwar als Summe aus dem Inhalt eines Basisregisters (BX oder BP), dem eines Indexregisters (DI oder SI) und beliebig vielen Konstanten. Die Konstanten können auch Variablennamen sein. Die allgemeine Form der indirekten Adressierung in 16-Bit-Umgebungen ist:

[Basisregister + Indexregister + Konstanten]

Es dürfen auch ein oder zwei Anteile entfallen. (Wenn nur eine Konstante in den Klammern steht, ergibt sich eine direkte Adressierung.) Die eckigen Klammern sind Pflicht. Die möglichen Varianten sind also:

```
[ BX ]            [ BX + Konstante]
[ BP ]            [ BP + Konstante]
[ DI ]            [ DI + Konstante]
[ SI ]            [ SI + Konstante]
[ BX + DI ]       [ BX + DI + Konstante]
[ BX + SI ]       [ BX + SI + Konstante]
[ BP + DI ]       [ BP + DI + Konstante]
[ BP + SI ]       [ BP + SI + Konstante]
                  [ Konstante ]
```

Stehen innerhalb der eckigen Klammern mehrere Konstante, so werden sie schon bei der Übersetzung vom Assembler zusammengefasst. Beispiel:

```
inc [1+Haeufigkeit+30+5]
```

wird bei der Übersetzung zu

```
inc [Haeufigkeit+36]
```

Eine wichtige Frage ist: In welchem Segment wird zugegriffen? Dies ist durch die Bauart des Prozessors festgelegt. Es gilt:

- Der Prozessor greift im Stacksegment zu, wenn das Basisregister BP ist.
- Der Prozessor greift in allen anderen Fällen im Datensegment zu.

Zum Laden der beteiligten Register mit dem Offset einer Variablen kann der Operator `Offset` verwendet werden. So ergeben sich dann sehr viele Möglichkeiten die Adressierung aufzubauen. An einem kleinen Beispiel sei die Vielfalt demonstriert. Es soll das Zeichen Nr. 5 in einem Feld von Zeichen überschrieben werden.

```
.DATA\\
Zeichenkette DB 'ABCDEfGH'
.CODE
  mov ax,@data
  mov ds,ax

  mov [zeichenkette + 5],'F'    ;direkte Adressierung

  mov bx,5
  mov [zeichenkette + bx],'F'   ;indirekte Adressierung mit BX + Konst.

  mov bx,5
  mov [zeichenkette + di],'F'   ;indirekte Adressierung mit DI + Konst.

  mov bx,offset zeichenkette    ;Offset von zeichenkette nach BX
  mov [bx+5],'F'    ;indirekte Adressierung mit BX + Konstante

  mov bx,offset zeichenkette    ;Offset von zeichenkette nach bx
  mov si,5
  mov [bx+si],'F'               ;indirekte Adressierung mit BX+SI

  mov bx,offset zeichenkette    ;Offset von zeichenkette nach bx
  add bx,5                      ;BX um 5 erhöhen
  mov [bx],'F'                  ;indirekte Adressierung mit bx

  mov bx,offset zeichenkette    ;Offset von zeichenkette nach bx
  mov si,4
  mov [bx+si+1],'F'             ;indirekte Adressierung mit BX+SI+Konst.

  mov si,offset zeichenkette+5  ;Offset von zeichenkette+5 nach si
  mov [si],'F'                  ;indirekte Adressierung mit bx
```

Alle Adressierungen in diesem Beispiel adressieren die gleiche Speicherzelle! Man beachte, dass die Adressierungen mit BP bewusst vermieden wurden, da dies den Stack adressieren würde.

Die indirekte Adressierung gibt uns also die Möglichkeit, den Inhalt eines Registers als variablen Zeiger in Speicher zu benutzen.

Wichtig: Variable Zeiger lassen sich nur mit Registern realisieren!

Eine Konstruktion über Speichervariable, die als Zeiger wirken sollen ist nicht möglich. Beispiel:

```
.DATA
Zeichenkette DB 'ABCDEfGH'
Zeiger DW ?
.CODE
mov ax,@data
mov ds,ax

mov zeiger, offset zeichenkette     ;Offset von zeichenkette in zeiger
mov [zeiger+5],'F'                  ;ACHTUNG: FEHLER!!!
```

Dieses Programmstück wird klaglos übersetzt, funktioniert aber nicht so, wie es gedacht war. Bei der Übersetzung wird für den Bezeichner 'zeiger' der Offset dieser Variablen eingesetzt (8), in der eckigen Klammer steht also der konstante Ausdruck [8+5] also wird in dieser Zeile fest Speicherzelle 13 adressiert!

Typoperatoren

Ein Problem bleibt noch: Der 8086 kann bei einem Speicherzugriff 8 Bit ansprechen (Bytezugriff) oder 16 Bit (Wortzugriff). Der 386 kann sogar in einem Schritt auf ein Doppelwort mit 32 Bit zugreifen. Wenn der Speicher nun unter Verwendung eines Variablennamens adressiert wird, ist durch die Definition der Variablen die Art des Zugriffs festgelegt. Wird dagegen ein Registerinhalt als Adresse benutzt, ist evtl. der Assembler nicht in der Lage, die Art des Zugriffs zu bestimmen. Beispiel:

```
.DATA
Zaehler DB (0)
Spalte  DW ?
Feld DB 10 DUP(?)
.CODE
.
.
  inc Zaehler          ;Bytezugriff wegen Variablendeklaration
  dec Spalte           ;Wortzugriff wegen Variablendeklaration
  mov bx,offset Feld
  mov al,[bx]          ;Aus Zielregister AL erkannt: Bytezugriff
  inc [bx]             ;Unklar ob Byte- oder Wortzugriff!!
                       ;Assembler codiert Wortzugriff und gibt Warning aus
```

Diese Unklarheit wird beseitigt durch die Verwendung eines *Typoperators* .

```
inc BYTE PTR [bx]    ; Bytezugriff
```

Die erlaubten Typoperatoren sind:

BYTE PTR Auf die adressierte Speicherstelle wird als 8-Bit-Dateneinheit (Byte) zugeriffen.
WORD PTR Auf die adressierte Speicherstelle wird als 16-Bit-Dateneinheit (2 Byte, ein Wort) zugeriffen.
DWORD PTR Auf die adressierte Speicherstelle wird als 32-Bit-Dateneinheit (4 Byte, ein Doppelwort) zugeriffen.

In dem folgenden Beispiel wird der Typoperator BYTE PTR ausgenutzt um auf die beiden Bytes eines Wortes getrennt zuzugreifen

```
.DATA
Zaehler  DW ?
.CODE
.
.
  mov al,BYTE PTR [Zaehler]     ; niederwertiges Byte laden
  mov bl,BYTE PTR [Zaehler+1]   ; höherwertiges Byte laden
```

Nun sei noch das Beispiel mit der Bestimmung der Häufigkeit der Buchstaben in einer Zeichenkette vollständig angegeben.

```
  .MODEL SMALL     ; Speichermodell "SMALL"
  .STACK 100h      ; 256 Byte Stack reservieren
  .DATA
  Zeichenkette DB 'Morgenstund hat Gold im Mund',0
;  Zeichenkette DB 'AAABBC',0       ; zum Testen
  Haeufigkeit DB 256 DUP (0)
  .CODE
Programmstart:
  mov  ax,@data
  mov  ds,ax       ; Laufzeitadresse des Datensgments nach DS

  mov di,offset zeichenkette    ;indirekte Adressierung vorbereiten
                            ; wegen SMALL: Verw. von NEAR-Zeigern
Zeichenholen:
  mov bl,[di]               ;indirekte Adressierung der Zeichenkette mit DI
                            ;ein Zeichen aus der Kette nach bl laden
  mov bh,0                  ;indirekte Adressierung mit BX vorbreiten
  inc [Haeufigkeit + bx]    ;Adresse wird zusammengesetzt aus Startadresse Haeufigkeit
                            ;und dem Inhalt des Registers BX
                            ;Beispiel: Das gelesene Zeichen war ein 'A' (Code: 65)
                            ;Bx enthält jetzt den Wert 65 und es wird der
                            ;Speicherplatz [Haeufigkeit+65] indirekt adressiert
  inc di                    ;Zeiger auf nächstes Zeichen weiterruecken
  cmp bl,0                  ;Ende der Zeichenkette? Begrenzungszeichen ist 0.
  jne Zeichenholen          ;Wenn Zeichen nicht gleich 0 nächstes Zeichen einlesen
```

```
mov  ah,04Ch
int  21h                  ;Programm beenden
END Programmstart         ;Ende der Übersetzung
```

2.4.5 Die indirekte Adressierung beim i80386

Ab dem 80386 kann zusätzlich jedes der acht 32-Bit-Allzweckregister als Basisregister dienen und, außer dem ESP-Register, auch jedes als Indexregister. Beispiele:

```
mov [eax+ecx+10],edx
inc dword ptr[edx]
```

Man hat also nun fast unbegrenzte Freiheit bei der Adressierung, wenn man die 32-Bit-Register benutzt. Eine Adressierung mit z.B. `[cx]` ist nach wie vor nicht möglich. Ausserdem ist zu beachten, dass damit ein 32-Bit-Offset übergeben wird, der nur in einem 32-Bit-Segment einen Sinn ergibt. Eine weitere sehr nützliche Sache ist die sog. *Index-Skalierung* . Dabei kann der Inhalt des verwendeten Indexregisters bei der Adressierung mit den Faktoren 2, 4 oder 8 multipliziert werden. Dazu wird innerhalb der eckigen Klammern *2, *4 oder *8 hinzugefügt. Dies ist sehr praktisch bei der Adressierung von Wort oder Doppelwortfeldern, bei denen jedes Feldelement 2 bzw. 4 Byte belegt. Zum Beispiel ersetzt der Befehl

```
mov AX,[Wortfeld + ecx*2]
```

die Befehlsfolge

```
shl ecx,1
mov AX,[Wortfeld + ecx]
shr ecx,1
```

Man muss allerdings darauf achten, dass 16-Bit- und 32-Bit-Adressierung nicht ohne weiteres gemischt werden dürfen. In einem 32-Bit-Code müssen alle Adressen immer 32-Bit umfassen, z.B. [ebx+...]. Im 16-Bit-Code müssen immer 16 Bit-Adressen verwendet werden, z.B. [esi+...].

2.5 Testfragen

1. Überlegen Sie ob die folgenden Befehle korrekt sind:

```
.DATA
Zaehler1 DB ?
Zaehler2 DB 0
Endechar DB ?
Startchar DB 'A'
```

```
Pixelx DW ?
Pixely DW 01FFh
Schluessel DD 1200h
.CODE
mov Zaehler1, 100h   ;
mov Zaehler2, ax     ;

mov ah,2
mov dx, Startchar    ;
int 21h

movzx Endechar,'Q'   ;
mov edx,Startchar    ;
xchg Pixely,cx       ;

mov schluessel,ebp   ;
mov Pixelx,Pixely    ;
```

2. Überlegen Sie welche der folgenden Befehle zu Fehlermeldungen, Warnungen oder Laufzeitfehlern führen:

```
.DATA
Feld DB 25 DUP (0)
zahl DW 0
.CODE
.386
mov [Feld+cx],al

mov [Feld+ecx],al

mov al,[Feld+bl]

mov [bx+bp+10],0

mov [si+di+1],10h

mov bx, offset zahl
mov cl, [Feld+bx]

mov cl,Feld

inc [bx]
```

3. Wie ist der Inhalt der Register ax,cx,edx,esi nach der Ausführung der folgenden Befehle?

```
.DATA
Bytevar       DB 66h
wfeld        DW 711h,822h,933h
```

```
dwort       DD 12001300h
.CODE
mov bx,offset bytevar
mov ax,[bx]
mov di,2
mov cx,[bx+di]
inc di
mov edx,[bx+di]
add di,3
mov esi,[bx+di]
```

Antworten auf Seite 125.

Kapitel 3

Daten transportieren

3.1 Daten gleicher Bitbreite kopieren - MOV

Der Hauptbefehl um Daten zu transportieren ist der MOV-Befehl. Er ist das Arbeitspferd und wahrscheinlich der häufigste Befehl in 80x86-Programmen. Der MOV-Befehl kopiert eigentlich die Daten, denn er läßt die Daten im Quellbereich unverändert. Die Syntax des MOV-Befehles ist:

`MOV` *Zieloperand, Quelloperand*

Ziel- und Quelloperanden können Register, Speicherplätze oder Direktwerte (Konstanten) mit einer Breite von 8, 16 oder 32 Bit sein. Wichtig ist, dass die Bitbreite beider Operanden übereinstimmt. Ein Befehl wie `MOV EAX,BX` führt zu einer Fehlermeldung wie „Operand types do not match“, weil EAX ein 32-Bit-Register ist und BX ein 16-Bit Register. Der MOV-Befehl hat einige weitere Einschränkungen:

- Es können nicht beide Operanden Segmentregister sein.
- Direktoperanden können nicht in Segmentregister geschrieben werden
- Es können nicht beide Operanden Speicheroperanden sein.

Die letzte Einschränkung gilt übrigens für alle 80x86-Befehle. [1] Die Möglichkeiten des MOV-Befehles sind in Abb. 4 grafisch dargestellt.

Bei Direktoperanden erweitert der Assembler in MOV-Befehlen bei der Übersetzung automatisch die Bitbreite passend für den Zieloperanden mit Nullbits. So wird z.B. aus dem Befehl `mov, ax,1` im Maschinencode der Befehl `mov ax,0001h`.

Operationen, die nicht in einem MOV-Befehl ausgeführt werden können, müssen in zwei Schritten erledigt werden, z.B. der Transport eines Direktwertes in ein Segmentregister:

[1] Nur die Stringbefehle (s.Kap.16.9) können zwei Speicheroperanden ansprechen, diese werden dann aber nicht als Operanden genannt.

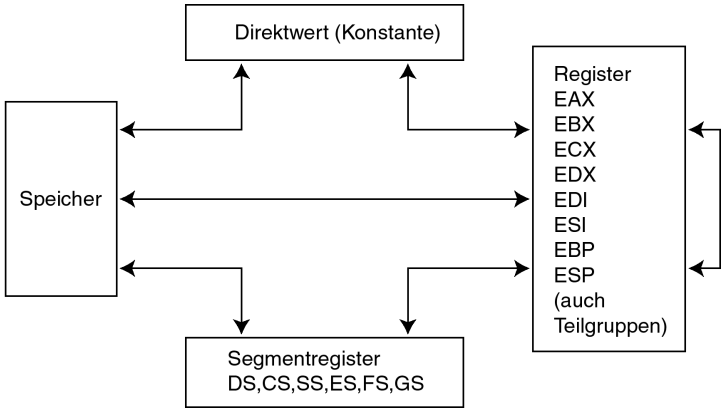


Abbildung 3.1: Möglichkeiten des MOV-Befehls. Ziel- und Quelloperand müssen gleiche Bitbreite haben.

```
mov ax,200h
mov es,ax
```

Eine ähnliche Sequenz findet man am Anfang des obigen Beispielprogrammes (und am Anfang aller anderen Programme).

3.2 Daten austauschen - XCHG

Mit dem Befehl XCHG, Exchange, können Daten zwischen zwei Operanden ausgetauscht werden, z.B. zwischen zwei Registern oder zwischen Register und Speicher. Die Syntax des Befehles ist:

`XCHG` *Operand1, Operand2*

So kann z.B. der Inhalt der Register EAX und EDX durch

```
xchg eax,edx
```

ausgetauscht werden. Ohne den XCHG-Befehl brauchte man dazu drei MOV-Befehle:

```
mov ebx,eax
mov eax,edx
mov edx,ebx
```

Diese Lösung hätte ausserdem den unerwünschten Nebeneffekt der Veränderung des Hilfsregisters EBX.

3.3 Daten in größere Register transportieren

Es kommt oft vor, dass Daten in ein Zielregister transportiert werden müssen, das mehr Bit hat als der Operand. Nehmen wir z.B. den Fall, dass eine vorzeichenlose 8-Bit-Zahl aus dem

Register dl in das Register AX übertragen werden soll. Der Befehl `mov ax,dl` führt nur zu einer Fehlermeldung. Es ist ja unklar, auf welche 8 Bit im Zielregister AX geschrieben werden soll. Man könnte sich nun z.B. entscheiden, auf die niederwertigen 8 Bit zu schreiben, dies müßte dann mit `mov al,dl` geschehen. Dann kann aber in AH noch ein Bitmuster stehen und der Inhalt von AX würde dann eine ganz andere Zahl repräsentieren. Man muss also zusätzlich die höherwertigen 8 Bit mit Null beschreiben:

```
mov al,dl
mov ah,0
```

Bei der Übertragung eines 8-Bit-Wertes in ein 32-Bit-Register würde man wie folgt vorgehen:

```
mov eax,0
mov al,dl
```

Es gibt nun einen Spezialbefehl, der diese Übertragung in einem Schritt durchführt: *MOVZX* , Move and extend Zero Sign, also Transportiere und erweitere mit Null-Bits.

Die obigen Operationen könnten also jeweils in einem Schritt durchgeführt werden:

```
movzx ax,dl
```

beziehungsweise

```
movzx eax,dl
```

Komplizierter wird die Situation, wenn die zu übertragende Zahl vorzeichenbehaftet ist, also im Zweierkomplement dargestellt ist. Dann müssen die höherwertigen Bit mit Null-Bits beschrieben werden, wenn die Zahl positiv ist und mit Eins-Bits wenn die Zahl negativ ist! Man müßte also zunächst das Vorzeichen ermitteln (Wie überhaupt?) und danach verzweigen, eine Sequenz von insgesamt mindestens fünf Befehlen:

```
                add dl,0            ; Vorzeichenflag setzen
                js negativ          ; jump if sign negativ
                mov eax,0           ; pos. Zahl, Nullbits schreiben
                jmp transport
negativ:        mov eax,0FFFFFFFFh  ; Eins-Bits schreiben
transport:      mov al,dl
```

Hier hilft der Befehl *MOVSX* , Move and extend sign, also Transportiere und erweitere mit Vorzeichen. Die obige Aufgabe kann dann mit einem Befehl erledigt werden:

```
movsx eax,dl
```

Die beiden Befehle MOVZX und MOVSX stehen erst ab dem 80386 zur Verfügung.

3.4 Bedingtes Setzen von Registern oder Speicherplätzen

Mit den Befehlen der SETcc-Familie, SET if Condition, kann abhängig von den Bedingungsflags eine 1 oder 0 in einen Zieloperanden geschrieben werden. Der Zieloperand muss ein 8-Bit-Register oder eine 8-Bit-Speichervariable sein. Die 1 oder 0 wird dann als 8-Bit-Wert (00h/01h) eingeschrieben. Die Bedingungsflags müssen zuvor durch einen CMP- oder SUB-Befehl gesetzt werden. Ein Beispiel:

```
cmp ax,bx            ; Compare ax,bx
setne dl             ; schreibt in dl eine 1 wenn ax ungleich bx ist
                     ; bzw. eine 0, wenn ax=bx
```

3.5 Testfragen

1. Entdecken Sie im folgenden Codeabschnitt die fehlerhaften Befehle:

```
 1:  mov al,50h
 2:  mov al,100h
 3:  mov 22,bh
 4:  mov cx,70000o
 5:  mov cx,70000
 6:  mov bx, 10001111000000b
 7:  mov eax,177FFA001h
 8:  mov edx, 02A4h
 9:  xchg cx,10h
10:  mov eax,-1
11:  mov eax,edi
12:  mov ah,bl
13:  mov bx,bl
14:  xchg eax,bp
15:  xchg dx,dx
16:  mov dl,di
17:  mov bp,bh
18:  xchg edi,dl
19:  mov esi,dx
20:  xchg esi,ebx
21:  xchg ch,cx
22:  mov ch,cl
```

2. Bestimmen Sie den Inhalt des Registers EAX nach der folgenden Befehlssequenz:

```
mov bx, 7856h
xchg bl,bh
mov ax, 3412h
xchg al,ah
shl eax,16      ; Inhalt von eax um 16 Bit nach links schieben
```

```
                    ; rechts werden Null-Bits nachgezogen
mov ax,bx
```

3. Vereinfachen Sie den folgenden Codeabschnitt:

```
1:  mov al,0
2:  mov ah,1

3:  mov ebx,0
4:  mov bx,2800h

5:  mov eax,0
6:  mov al,dl

7:  xchg ax,ax

8:  mov ax,si
9:  mov si,di
10: mov di,ax
```

4. Es soll folgende Aufgabe (ein Ringtausch) bewältigt werden:
   – Inhalt von AX nach BX bringen
   – Inhalt von BX nach CX bringen
   – Inhalt von CX nach AX bringen
   Dabei sollen natürlich keine Daten verloren gehen! Schreiben sie Befehlssequenzen um die Aufgabe zu lösen:
   a) mit mov-Befehlen und b) kürzer! (Wie?)

5. Schreiben sie jeweils eine Befehlssequenz um folgendes zu bewirken:
   a) höherwertiges Wort von EAX nach DI bringen und niederwertiges Wort von EAX nach SI bringen
   b) CX ins niederwertige Wort von EAX bringen und DX ins höherwertige Wort von EAX bringen
   c) CL ins niederwertige Byte von DX bringen und CH ins höherwertige Byte von DX bringen
   Hierbei müssen auch die shift-Befehle shl und shr benutzt werden.

6. Setzen Sie mit *einem* Transportbefehl das höherwertige Wort von EAX gleich Null, ohne das niederwertige Wort zu verändern!

Lösungen auf Seite 126.

Kapitel 4

Ein- und Ausgabe

Der Mikroprozessor tauscht nicht nur mit dem Hauptspeicher Daten aus, sondern auch mit der Aussenwelt und anderen Hardwarebausteinen. Ein einfaches Beispiel ist die Tastatur: Wenn der Benutzer eine Taste drückt, erzeugt die Tastatur einen Code (den Scancode). Die Tastatur legt diesen Code an einen sog. *Eingabebaustein* , der am Bussystem des Computers angeschlossen ist. Der Prozessor liest die Daten von diesem Eingabebaustein. In anderen Fällen müssen Daten an die Aussenwelt, z.B. einen Drucker, übergeben werden. Dazu wird ein *Ausgabebaustein* benutzt. Der Prozessor schreibt die Daten auf den Ausgabebaustein und erteilt dem Ausgabebaustein eine Freigabe, die Daten an das angeschlossene Gerät weiterzugeben. Die Ein- und Ausgabebausteine haben Adressen genau wie Speicherplätze, allerdings ist der Adressraum kleiner.

Man nennt die beiden Vorgänge auch *Eingabe* und *Ausgabe* , engl. *Input* und *Output* . Der Mikroprozessor hat dazu die beiden Maschinen- bzw. Assemblerbefehle IN und OUT, abgekürzt auch I/O. Die Ein-/Ausgabebausteine nennt man auch *I/O-Ports* und ihre Adressen *I/O-Portadressen* . Durch die Verwendung von IN und OUT ist sichergestellt, dass nicht auf den Speicher sondern auf die I/O-Ports zugegriffen wird. Da Ein- und Ausgaben viel seltener sind als Hauptspeicherzugriffe, hat man hierbei viel weniger Komfort als bei letzteren.

Für beide Befehle muss die I/O-Portadresse im Register DX hinterlegt werden. Ist diese Adresse allerdings kleiner als 100h (also max. FFh), so kann sie als Direktoperand im IN- oder OUT-Befehl genannt werden. Der IN- und OUT-Befehl kann in 8-, 16- oder 32-Bit Breite ausgeführt werden. Ziel bzw. Quellregister ist AL, AX oder EAX je nach Bitbreite.

Beispiele:

```
1. Senden eines Zeichens über die serielle Schnittstelle COM1
    mov dx,3F8h     ; IO-Portadresse von COM1, größer als FFh
    out dx,al       ; Byte in AL am COM1-Baustein übergeben (wird gesendet)

2.  Einlesen der Interrupt Enable Mask vom Interrupt-Controller
    in al, 20h      ; IO-Adresse des Interruptcontrollers ist 20h,
                    ; also kleiner als FFh
```

Kapitel 5

Betriebssystemaufrufe

5.1 Allgemeines

Die Assemblersprache verfügt - im Gegensatz zu Hochsprachen - nicht über komplexe Befehle um Bildschirmausgaben, Dateizugriffe, Bedienung von Schnittstellen u.a.m. durchzuführen. Wenn man dabei direkt auf die Hardware zugreifen wollte hätte man große Probleme: Man brauchte sehr gute Hardwarekenntnisse, die Programme wären extrem aufwendig und vor allem hardwareabhängig. So würden viele Programme auf neueren Rechnern nicht mehr laufen. In solchen Fällen muß in Assemblerprogrammen ein *Betriebssystemaufruf* , kurz Systemaufruf, durchgeführt werden. Solche Betriebssystemaufrufe kommen daher im Ablauf fast aller Assemblerprogramme vor, zumindest jedoch am Programmende, wo die Kontrolle mit einem Systemaufruf wieder an das Betriebssystem zurückgegeben wird.[1]

Wir wollen die Verhältnisse am Beispiel eines PC unter DOS näher betrachten. Der unmittelbare Zugriff auf die Hardware-Komponenten erfolgt über das sog. Basic Input/Output System, das *BIOS* . Das BIOS ist eine Sammlung von Unterprogrammen um die Bausteine des Rechners direkt anzusteuern, z.B. die Grafikkarte, die Laufwerke, die Schnittstellen, den Speicher, die Uhr u.a.m. Das BIOS ist in einem EPROM gespeichert, einem Festwertspeicher auf der Hauptplatine. Es ist nach dem Einschalten des PC sofort verfügbar.

DOS steht für Disk Operating System und ist z.B. verfügbar als MS-DOS oder PC-DOS. DOS ist das eigentliche Betriebssystem, es übernimmt die Aufgabe Programme zu starten und zu stoppen, Laufwerke und Dateisysteme zu verwalten, Geräte zu steuern, Treiber einzubinden u.a.m. DOS nimmt dazu Funktionen des BIOS in Anspruch.

Ebenfalls möglich sind direkte Hardwarezugriffe über I/O–Adressen oder direkt adressierte Speicherplätzen. Dies sollte aber eigentlich der Systemprogrammierung vorbehalten sein und bei Anwenderprogrammen eine Ausnahme bleiben. Einen Überblick gibt Abb. 5.1.

Wie sind nun BIOS und DOS realisiert und welche Schnittstellen bieten sie dem Anwendungsprogrammierer? Die Intel–Prozessoren unterstützen hardwaremäßig einen Satz von sehr systemnahen Unterprogrammen, die nicht über Adressen sondern über Nummern zwischen 0 und 255

[1] In Hochsprachenprogrammen werden diese Systemaufrufe ebenfalls durchgeführt. Sie bleiben allerdings meist unbemerkt, weil der Compiler die notwendigen Aufrufe automatisch erzeugt.

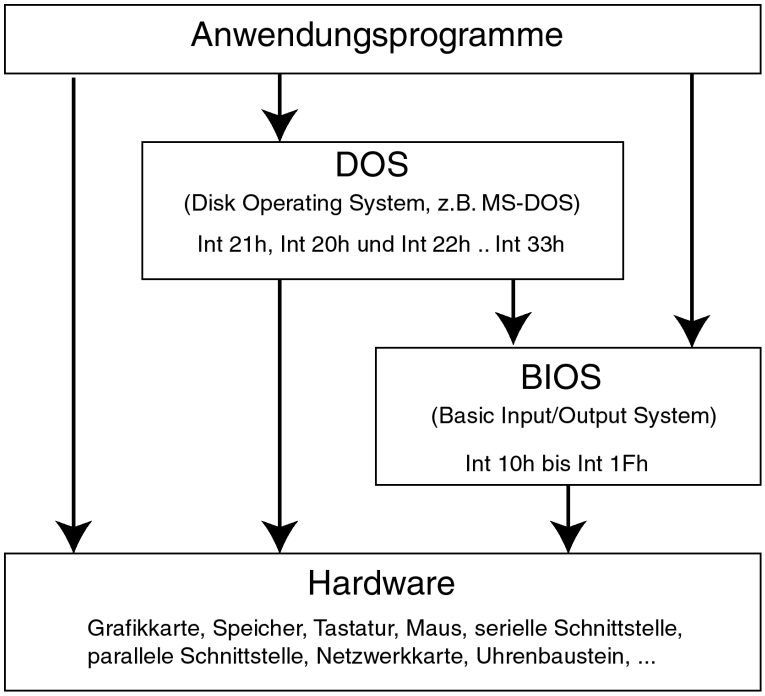


Abbildung 5.1: Betriebssystemaufrufe und Ansteuerung der Hardware unter DOS

aufgerufen werden: die *Interrupts*. Die Adressen dieser Unterprogramme stehen in der sog. *Interrupt–Vektoren–Tabelle* und werden beim Aufruf eines Interrupts automatisch vom Prozessor geladen. Die Nummern der Interrupts werden üblicherweise hexadezimal genannt. Die Interrupts können in fünf Gruppen eingeteilt werden.

1. Interrupts die der Prozessor selbst in Ausnahmesituationen auslöst, die sog. *Exceptions* bzw. Ausnahmen.

2. Interrupts, die durch externe Ereignisse hardwaremäßig ausgelöst werden und das laufende Programm unterbrechen. (→ Name)

3. BIOS–Aufrufe

4. DOS–Aufrufe

5. Interrupts die frei bleiben und durch die Anwendung, d.h. das eigene Programm, belegt werden können.

In der Assemblerprogrammierung werden die Aufrufe der dritten und vierten Gruppe häufig benutzt. Die Belegung der Interrupts ist teilweise vom Typ des Rechners bzw. der Betriebssystemversion abhängig. In der folgenden Tabelle ist ein Auszug gegeben:

Nr.	Bedeutung	Gruppe
0	Division durch Null	Exc.
1	Einzelschrittbetrieb	Exc.
2	NMI	Exc.
3	Breakpoint	Exc.
4	Overflowauswertung mit INTO	Exc.
5	Bound / Print Screen	Exc./BIOS
6	ungültiger Opcode	Exc.
7	kein Coprozessor vorhanden	Exc.
8	IRQ0 Timerbaustein, 18.2 mal pro sec	HW
9	IRQ1 Tastatur	HW
Ah	IRQ2 kaskadierter 8259 IRQ8–IRQ15	HW
Bh	IRQ3 COM2	HW
Ch	IRQ4 COM1	HW
Dh	IRQ5 LPT2	HW
Eh	IRQ6 Diskettenlaufwerk	HW
Fh	IRQ7 LPT1	HW
10h	Videotreiber	BIOS
11h	Ermittlung der Systemkonfiguration	BIOS
12h	Speichergröße ermitteln	BIOS
.	.	BIOS
14h	Serielle Schnittstelle	BIOS
15h	Extended Memory–Zusatzfunktionen	BIOS
16h	Tastatur	BIOS
17h	Drucker	BIOS
.	.	BIOS
1Ah	Systemuhr	BIOS
.	.	BIOS
1Fh	Adresse der Grafikzeichensatztabelle	BIOS
20h	Programmende	DOS
21h	DOS–Funktionsaufruf, Funktionsnummer in AH übergeben	DOS
.	.	DOS
27h	Programm beenden und resident halten (TSR)	DOS
.	.	DOS
2Fh	CD–Rom	DOS
.	.	DOS
33h	Maus	DOS
60h	für eigene Anwendungen	User
.	.	User
67h	für eigene Anwendungen	User
70h	IRQ8 Echtzeituhr	HW
.	.	HW
75h	IRQ13 Coprozessor	HW
77h	IRQ15 reserviert	HW

Die größte Gruppe stellt hierin Int 21h, der DOS-Funktionsaufruf dar. Das System der Interrupts hat für den Programmierer mehrere Vorteile: Man braucht sich bei Betriebssystem–Aufrufen nicht um Adressen zu kümmern, die Nummer genügt. Diese bleibt richtig, auch wenn eine neue

Betriebssystemversion installiert wird. Andererseits können Interruptvektoren geändert werden und z.B. auf eigene Routinen zeigen. Damit es ist z.B. möglich auf Ereignisse in der Hardware mit eigenen Programmstücken zu reagieren. Außerdem gibt es freie Interrupts, d.h. dass der Satz von Betriebssystem–Aufrufen durch eigene Routinen erweitert werden kann, die dann in jeder beliebigen Programmiersprache leicht erreichbar sind!

5.2 Ausführung von Betriebssystemaufrufen in Assembler

Die Ausführung von Betriebssystemaufrufen in Assembler geschieht in drei Phasen:

1. **Vorbereitung:**
   A) Die nähere Spezifizierung des gewünschten Aufrufes durch Ablage der entsprechenden Funktionsnummern in Registern. Dabei wird zumindest das Register AH belegt, evtl. außerdem AL.
   B) Soweit notwendig, die Übergabe von Parametern. Dies geschieht ebenfalls in Registern.
2. **Aufruf:**
   In jedem Falle der Befehl INT *Nr.* (Interrupt). Die Nummer variiert entsprechend dem gewünschten Aufruf (s.Tabelle).
3. **Auswertung:**
   Der Betriebssystemaufruf gibt in vielen Fällen Daten zurück, welche dann nach der Abarbeitung des BS-Aufrufs in Registern liegen. Diese können nun dort abgeholt und ausgewertet werden. Die Daten liegen meist in den Registern BX, CX, DX. Ein nach dem Aufruf gesetztes Carry–Flag zeigt einen Fehler an, die Fehlernummer findet sich in AL.

Zu den Betriebssystemaufrufen gibt es natürlich Dokumentationen, in denen die Schnittstelle, d.h. die Übergabe der Daten in Registern genau beschrieben ist. Dabei wird unterschieden zwischen

Datenübergabe vom Programm an das Betriebssystem (Vorbereitung)
Bezeichnung auch: *Daten vor Aufruf* oder *Eingabe*.

Datenübergabe vom Betriebssystem an das Programm (Auswertung)
Bezeichnung auch: *Daten nach Aufruf* oder *Rückgabe*.

Beispiele

In der Dokumentation findet man zu Funktion 1 von Int 21h z.B. folgende Information:

DOS-Funktion 01h : Zeichen von Tastatur einlesen

Vor Aufruf	AH=1
Nach Aufruf	Von Tastatur eingelesenes Zeichen in AL

Man muss also vor dem Aufruf in das Register AH eine 1 einschreiben, die Nummer des Funktionsaufrufes. Die aufgerufene Systemroutine übernimmt dann das Lesen des Zeichens von der Tastatur; sie wartet also geduldig, bis der Benutzer eine Taste drückt. Dann legt sie den Code

der gedrückten Taste in Register AL und beendet sich. Jetzt wird im Anwendungsprogramm der nach dem INT 21h folgende Befehl ausgeführt. Man wird hier normalerweise das in Al übergebene Zeichen auswerten. Programmbeispiel:

```
mov ah,1    ; DOS-Funktion 1 "Zeichen von der Tastatur lesen"
int 21h     ; DOS aufrufen
cmp al,13   ; compare, ist das eingelesene Zeichen der Code der Returntaste?
; usw.
```

Die DOS-Funktion 2 dient dazu, ein Zeichen auf den Bildschirm zu schreiben. In der Dokumentation findet man:

DOS-Funktion 02h : Zeichen auf Bildschirm ausgeben

Vor Aufruf	AH=2 DL=auszugebendes Zeichen
Nach Aufruf	---

Der folgende Programmabschnitt gibt ein Ausrufezeichen auf den Bildschirm aus:

```
mov ah,2    ; DOS-Funktion 2 "Zeichen auf Bildschirm ausgeben"
mov dl,'!'  ; auszugebendes Zeichen muss in DL liegen.
int 21h     ; DOS aufrufen
; keine Rückgabewerte, Auswertung entfällt
```

DOS und BIOS sind selbst Assemblerprogramme und benutzen die gleichen Register und Flags, wie unsere Anwenderprogramme. Sowohl DOS als auch BIOS sind in der Regel so programmiert, daß sie nur die als Ausgaberegister dokumentierten Register verändern. Eine Ausnahme bildet AX, das oft verändert wird. Für die Programmierung von Betriebssystemaufrufen verwendet man meist Tabellen, in denen die Belegung der Register vor und nach dem jeweiligen Aufruf dokumentiert ist.

5.3 Einige Nützliche Betriebssystemaufrufe

DOS, INT21h, Funktion 01h : Zeichen von Tastatur einlesen

Vor Aufruf	AH=1
Nach Aufruf	Von Tastatur eingelesenes Zeichen in AL

DOS, INT21h, Funktion 02h : Zeichen auf Bildschirm ausgeben

Vor Aufruf	AH=2 DL=auszugebendes Zeichen
Nach Aufruf	—

DOS, INT21h, Funktion 4Eh: Finde ersten passenden Verzeichniseintrag

Vor Aufruf	AH=4Eh DS:DX = Zeiger auf den Offset eines ASCIIZ-Strings mit der Verzeichnis- bzw. Suchmaske, hier '*.ASM',0 CX = Attribute, hier 0
Nach Aufruf	AX = 0 wenn fehlerfrei, AX = Fehlercode, wenn Fehler, z.B. keine passenden Einträge CF = 1, wenn Fehler

DOS, INT21h, Funktion 4Fh: Finde weiteren passenden Verzeichniseintrag

Vor Aufruf	AH=4Fh
Nach Aufruf	AX = 0 wenn fehlerfrei, AX = Fehlercode, wenn Fehler, z.B. keine weiteren Einträge CF = 1, wenn Fehler

DOS, INT21h, Funktion 2Fh: Ermittle DTA-Adresse

Vor Aufruf	AH=2Fh
Nach Aufruf	BX = Offset des Zeigers auf die DTA ES = Segment des Zeigers auf die DTA

DOS, INT21h, Funktion 30h : DOS-Versionsnummer ermitteln

Vor Aufruf	AH=30h
Nach Aufruf	AL = Hauptversionsnummer AH = Nebenversionsnummer

DOS, INT21h, Funktion 36h: Ermittlung der Diskettenkapazität

Vor Aufruf	AH=36h DL = logische Laufwerksnummer, A=1, B=2, usw.
Nach Aufruf	AX=Sektoren pro Cluster BX=Anzahl der freien Cluster CX = Anzahl der Bytes pro Sektor DX = Anzahl der Cluster insgesamt

BIOS, INT 16h, Funktion AH=02h, Tastatur-Flags ermitteln

Vor Aufruf	AH=02h
Nach Aufruf	AH = reserviert AL = Shift-Status-Byte

BIOS, INT 10h, Funktion 02h : Setzen der Cursor-Position

Vor Aufruf	AH=02h BH = Bildschirmseite, hier 0 DH = Reihe, oberste=0 DL = Spalte, links=0
Nach Aufruf	—

5.4 Testfragen

Jeder der folgenden fünf Codeabschnitte enthält einen Fehler; entdecken sie die Fehler!

```
   .DATA
   Meldung1  DB   'Ende des Beispielprogramms',13,10,'$'
   Meldung2  DB   'Programm beendet',13,10

    .CODE
Programmstart:           ; Label haben einen Doppelpunkt am Ende
   mov  ax,@data             ; Uebergabe der Adresse des Datensegments
                ; zur Laufzeit
   mov  ds,ax                ; DS zeigt nun auf das Datensegment

   ; Alle folgenden Abschnitte enthalten je einen Fehler, finden Sie diese!!

   ; *************   Abschnitt1   ****************
   mov  dx,OFFSET Meldung1   ; Offset der Adresse des Strings
   int  21h                  ; Interrupt 21h : Aufruf von DOS

   ; *************   Abschnitt2   ****************
   mov dl,'A'
   mov ah,02h            ; Bildschirmausgabe mit DOS
   int 21

   ; *************   Abschnitt3   ****************
   mov ah,03h       ; Funktion 3: Lies Cursorposition und -groesse
            ; Rueckgabe ch,cl: erste und letzte Scanlinie
            ;           dh,dl: Reihe und Spalte
   mov bx,0     ; Bildschirmseite 0
   int 10h      ; Int 10h (Video)
   mov cx,0
   mov dx,ax

   ; *************   Abschnitt4   ****************
   mov  ah,9                 ; DOS-Funktion, die einen durch $ begrenzten
                ; String auf den Bildschirm ausgibt
   mov  dx,OFFSET Meldung2   ; Offset der Adresse des Strings
   int  21h                  ; Interrupt 21h : Aufruf von DOS

   ; *************   Abschnitt5   ****************
; Programmende, die Kontrolle muss explizit an DOS zurueckgegeben werden
   mov  ah,04Ch              ; ah=04C : DOS-Funktion "terminate the program"
   mov  al,0             ; DOS-Return-Code 0
```

Lösungen auf Seite 127.

Kapitel 6

Bitverarbeitung

6.1 Bitweise logische Befehle

Die Befehle dieser Gruppe arbeiten bitweise parallel. Sie führen die Operationen des logischen UND, ODER und exclusiv ODER sowie der Invertierung zwischen zwei 8-, 16- oder 32-Bit-Operanden durch. Dabei werden die Bits des einen Operanden mit den entsprechenden Bits des anderen Operanden verknüpft und das Ergebnis landet im ersten Operanden. Die bitweise logischen Befehle setzen die Flags wie folgt:

ZF	gesetzt, wenn Ergebnis Null ist, sonst gelöscht
SF	gleich dem MSB des Ergebnis
PF	gesetzt, wenn die Parität des niederwertigen Byte gerade
CF,OF	immer gleich NUll

AND – das logische UND

Der AND-Befehl verknüpft zwei Operanden bitweise entsprechend dem logischen UND: Das Ergebnisbit ist gleich Eins, wenn beide Operandenbits gleich eins sind, sonst Null. Die Operanden können 8, 16 oder 32 Bit haben und Register-, Speicher- oder Direktoperanden sein. Da das Ergebnis im ersten Operanden abgelegt wird kann dieser kein Direktoperand sein. Ein Beispiel:

```
mov al, 0C3h    ; AL = 11000011b
and al, 66h     ; AND  01100110b
        ;Ergebnis AL = 01000010b = 42h
```

Der AND-Befehl ist nützlich um ausgewählte Bits eines Operanden zu löschen (auf Null zu setzen). Im folgenden Beispiel wird in AX Bit 3 gelöscht. In BX werden alle Bits außer Bit 6 gelöscht. Anschließend wird ein bedingter Sprungbefehl ausgeführt, falls Bit 6 gleich Null ist: verzweigung

```
and AX,1111111111110111b   ; oder auch: and ax, FFF7h
```

```
and BX,0000000001000000b   ; oder auch and bx, 40h
jz Bit6gleichNull          ; Sprung wird ausgeführt, wenn Ergebnis Null
                           ; Ergebnis Null, wenn Bit 6 gleich Null
```

TEST – eine nützliche Variante von AND

Will man mit dem AND-Befehl mehrere Bits prüfen, muss man den Operanden jeweils vorher sichern, weil er beim ersten AND schon verändert wird. Für diese Anwendung ist der Befehl TEST maßgeschneidert. Er arbeitet genau wie AND mit dem einen Unterschied, dass er das Ergebnis *nicht* in den ersten Operanden zurückschreibt. Die Flags werden aber wie bei AND gesetzt. Der Nutzen des TEST-Befehls liegt allein in der Auswertung dieser Flags. Im folgenden Beispiel wird nacheinander Bit 2 und Bit 4 von EAX ausgewertet ohne EAX zu verändern:

```
test eax,04h     ; Bit 2 gesetzt?
jnz Bit2gesetzt  ; jump if not zero ...
test eax,10h     ; Bit 4 gesetzt?
jz Bit4gesetzt   ; jump if zero ...
```

OR – das logische ODER

Ein bitweise logisches ODER wird durch den Befehl OR durchgeführt: Die Ergebnisbits sind nur dann gleich Null, wenn beide Operandenbits gleich Null sind, sonst Eins. Für die Operanden gilt das gleiche wie bei AND. OR ist geeignet, um ausgewählte Bits eines Operanden gleich eins zusetzen. Beispiel:

```
mov al, 0CCh    ; AL = 11001100b
or al, 2h       ; OR   00000010b
        ;Ergebnis AL = 11001110b = CEh
```

XOR – das exclusive ODER

Ein bitweise logisches exclusives ODER wird durch den XOR-Befehl durchgeführt: Ein Ergebnisbit ist gleich Eins, wenn die Operandenbits ungleich sind, sonst gleich Null. Operanden: wie OR. Beispiel:

```
mov al, 0C3h    ; AL = 11000011b
xor al, 033h    ; XOR  00110011b
        ;Ergebnis AL = 11110000b = 0
```

Der xor-Befehl kann benutzt werden um gezielt einzelne Bits zu invertieren (toggeln), z.B. xor ax,02h invertiert Bit 1 und läßt alle anderen Bits unverändert.

NOT – bitweise Invertierung

Der letzte Vertreter ist der NOT-Befehl, der einfach alle Bits eines Operanden invertiert und daher auch nur einen Operanden braucht. Beispiel:

```
mov al, 0E5     ;           AL = 11100101b
not al          ; Ergebnis: AL = 00011010b = 1Ah
```

Weitere Anwendungen

Die bitweise logischen Befehle werden manchmal etwas trickreich genutzt, z.B.

Schnelles Null-setzen Oft müssen Register auf den Wert Null gesetzt werden. Das geht auch mit:

```
xor ax,ax  ; Schneller und kürzer als mov ax,0
```

Schnelles Setzen von Flags Nehmen wir an, in Register AL befindet sich ein unbekannter Wert und wir wollen wissen ob dieser Wert Null ist. Das Zeroflag kann nun gesetzt werden mit

```
or al,al   ; kürzer und schneller als cmp al,0
```

6.2 Schiebe- und Rotationsbefehle

Die Befehle dieser Gruppe erlauben es, ein komplettes Bitmuster nach links oder rechts zu schieben. Dabei wird an einem Ende ein Bit „herausfallen'". Wenn dieses herausgefallene Bit am anderen Ende der Datenstruktur wieder eingesetzt wird spricht man von *Rotation* , sonst von *Schieben* (Shift). Das bearbeitete Bitmuster kann in einem Register oder im Hauptspeicher liegen und 8, 16 oder 32 Bit umfassen. Im folgenden Beispiel wird ein Bitmuster um ein Bit nach rechts geschoben.

```
mov al,11000110b
shr al,1         ; shift right al 1
                 ; in al steht nun 01100011b = 63h
```

Die Syntax umfasst immer zwei Operanden: Das zu bearbeitende Bitmuster und die Anzahl Bits die geschoben oder rotiert werden soll. Die Bitzahl kann eine Konstante sein oder in CL stehen. (Konstanten größer 1 erst ab 80286)

Schiebe-/Rotationsbefehl Reg/Mem, Konstante/CL

Im folgenden sollen die acht Varianten der Schiebe- und Rotationsbefehle kurz betrachtet werden. Allen gemeinsam ist, dass das letzte herausgefallene Bit ins Carryflag geschrieben wird.

SHL Shift Left, SHR Shift Right

Einfaches Schieben nach links oder rechts, die frei werdenden Bitstellen werden mit einer Null aufgefüllt. Für Binärzahlen gilt ja: Das einfache Schieben um ein Bit nach links entspricht einer Multiplikation mit zwei, nach rechts einer Division durch zwei. Man kann also mit den Schiebebefehlen sehr gut *2 und /2 rechnen, SHL und SHR funktioniert allerdings nur bei vorzeichenlosen Zahlen.

SAL Shift Arithmetic Left, SAR Shift Arithmetic Right

Speziell für das Rechnen durch Schieben ausgelegt! Leisten Division durch zwei oder Multiplikation mit zwei auch bei vorzeichenbehafteten Zahlen! SAL arbeitet exakt wie SHL. SAR dagegen funktioniert etwas ungewöhnlich: Beim Schieben nach rechts wird das MSB sowohl geschoben als auch auf das neue MSB reproduziert. Die folgenden Beispiele zeigen, dass damit auch für negative Zahlen richtig gerechnet wird.

```
; Beispiel: Multiplizieren mit 2
    mov al,-1   ; al=11111111b = -1
    sal al,1    ; al=11111110b = -2
    sal al,1    ; al=11111100b = -4
    ; usw.
```

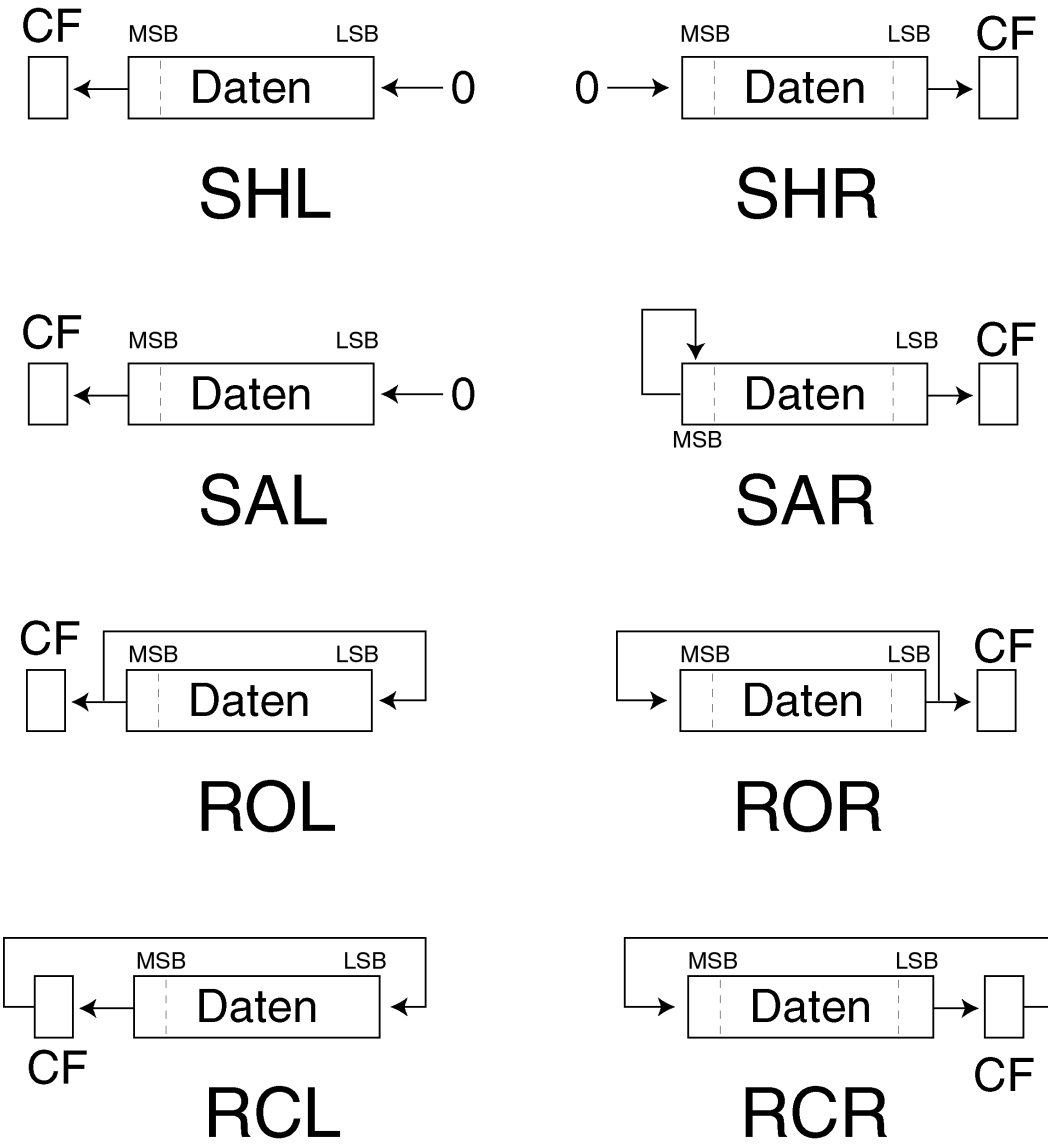
```
; Beispiel: Dividieren durch 2
    mov al,-16  ; al=11110000b = -16
    sar al,1    ; al=11111000b = -8
    sar al,1    ; al=11111100b = -4
    sar al,1    ; al=11111110b = -2
    sar al,1    ; al=11111111b = -1
    sar al,1    ; al=11111111b = -1, Rundung nach -unendlich
    ; usw.
```

ROL Rotate Left, ROR Rotate Right

Einfache Rotationen nach links oder rechts: Das herausgefallene Bit kommt auf die freiwerdende Bitstelle und ins Carryflag.

RCL Rotate through Carry Left, RCR Rotate through Carry Right

Ähnlich ROL und ROR, mit dem Unterschied, dass hier das Carryflag als Bit auf die freiwerdende Stelle gelangt und das herausgefallene Bit ins Carryflag kommt. Das Carryflag ist hier quasi ein Teil der Datenstruktur.



Beispiele

1. Übertragung eines 16-Bit-Wertes aus BX in das höherwertige Wort von EAX:

```
mov ax,bx
shl eax,16
```

2. Schnelle Multiplikation des Wertes in AX mit 9: [1]

```
mov bx,ax   ; Kopie anlegen
sal ax,3    ; ax = ax*8
add ax,bx   ; ax = ax*9
```

[1] Der Geschwindigkeitsvorteil bei Multiplikation durch Schiebebefehle gegenüber dem IMUL-Befehl hängt vom Prozessortyp ab und muss im Einzelfall anhand der nötigen Taktzyklen nachgerechnet werden.

3. Entnehmen eines Zahlenwertes auf einem Bitfeld innerhalb eines Bytes.

6.3 Einzelbit-Befehle

Ab dem 80386-Prozessor gibt es spezielle Befehle um einzelne Bits abzufragen oder zu ändern. Bei diesen Befehlen ist es nicht mehr nötig, mit Masken zu arbeiten, um auf einzelne Bits zuzugreifen.

BT – Bits testen

Mit dem Befehl BT wird ein einzelnes Bit eines Register- oder Speicheroperanden in das Carryflag übertragen. Im folgenden Beispiel wird Bit 3 von EDX getestet:

```
bt edx,3        ; Bit 3 ins CF übertragen
jc bit3gesetzt  ; Auswertung des Bits durch bedingten Sprung
```

Es gibt drei Varianten des BT-Befehls, der zusätzlich das getestete Bits anschließend verändert:

BTS (Bit Test and Set) Setzt das getestete Bit auf 1

BTR (Bit Test and Reset) Setzt das getestete Bit auf 0

BTC (Bit Test and Complement) Komplementiert (invertier) das getestete Bit

BSF und BSR – 1-Bits suchen

Mit den Befehlen BSF (Bit search forward) und BSR (Bit search reverse) wird in einem Operanden das erste 1-Bit gesucht. BSF beginnt die Suche bei Bit 0 (LSB), BSR beginnt beim MSB. Sind alle Bits Null wird das Zeroflag gelöscht. Wenn nicht, wird das Zeroflag gesetzt und die gefundene Bitposition im Zielregister gespeichert.

```
mov dx, 0100001101101100b
bsf cx,dx       ; Bit 2 ist erstes 1-Bit, -> cx=2
```

6.4 Testfragen

1. Bestimmen Sie den Inhalt der Register AX, BX, CX, DX nach der Ausführung der folgenden Befehle:

```
mov ax,0FEDCh
and ax, 1234h
mov bx, 1234h
or  bx, 4321h
mov cx, 6789h
xor cx, 9876h
mov dx, 12EFh
not dx
```

2. Bestimmen Sie den Inhalt der Register AL, BL, CL, DL nach der Ausführung der folgenden Befehle:

```
mov al,95h
shl al,2
mov bl,95h
sar bl,1
mov cl,95h
rol cl,2
mov dl,95h
sal dl,1
rcl dl,1
```

3. Schreiben Sie eine Programmsequenz, die folgende Bitoperationen an Register AX vornimmt:

   - Bit 0 und Bit 5 löschen
   - Bit 1 und Bit 3 setzen
   - Bit 2 und Bit 7 invertieren
   - alle anderen Bits unverändert lassen

   Formulieren Sie die Konstanten binär und hexadezimal!

4. Führen Sie nur mit Bitbefehlen sowie `mov` und `add` folgende Berechnung aus: AX = 17*BX + 9*CX . Ein Registerüberlauf soll zunächst nicht betrachtet werden.

5. Schreiben Sie ein Programmstück, das nur mit Bitbefehlen sowie `add` und evtl. `mov` arbeitet und folgendes ausführt: Der Inhalt von ax soll um eins erhöht werden, wenn in Register BX Bit 7=0 ist!

Antworten auf Seite 128.

Kapitel 7

Sprungbefehle

Sprungbefehle sind in Assemblersprache von elementarer Bedeutung, weil Verzweigungen und Wiederholungen mit Sprungbefehlen realisiert werden. Man unterscheidet unbedingte Sprungbefehle, bedingte Sprungbefehle und Unterprogrammaufrufe; letztere werden in einem eigenen Kapitel behandelt.

7.1 Unbedingter Sprungbefehl - JMP

Mit dem Befehl JMP, Jump, Springe, wird ein unbedingter Sprung ausgeführt. Die Syntax ist

JMP Sprungziel

Das Sprungziel ist in der Regel eine Marke, die irgendwo im Programm erklärt ist. Beispiel:

```
Lese_neues_zeichen:
    .
    .
    jmp Lese_neues_zeichen      ;direkter unbedingter Sprung
```

Man spricht in diesem Fall auch vom *direkten Sprung* . Seltener wird der *indirekte Sprung* verwendet, bei dem das Sprungziel in einem Register oder sogar einem Speicherplatz liegt. Beispiel:

```
    mov ax, offset Sprungmarke5
    jmp ax                      ;indirekter unbedingter Sprung
```

Aus der Sicht des Programmierers braucht für unbedingte Sprünge immer nur der Befehl jmp verwendet werden. Auf Maschinencode-Ebene werden dagegen die Sprungbefehle nach der Sprungweite weiter unterschieden:

Entfernung des Sprungzieles	Bezeichnung	Befehlslänge im Maschinencode
-128 .. +127 Byte	SHORT JUMP	1 Byte OpCode + 1 Byte rel. Entfernung
im gleichen Segment	NEAR JUMP	1 Byte OpCode + 2 Byte NEAR-Zeiger
anderes Segment	FAR JUMP	1 Byte OpCode + 4 Byte FAR-Zeiger

Für die SHORT JUMP's ist die Sprungweite relativ codiert, 0 bedeutet, dass der folgende Befehl ausgeführt wird. Wenn das Sprungziel nah genug ist, können relative Sprünge erzwungen werden durch `jmp short ...`

7.2 Bedingte Sprungbefehle

Bedingte Sprungbefehle sind von Bedingungen abhängig. Ist die Bedingung wahr, wird der Sprung ausgeführt, ist sie falsch, wird er nicht ausgeführt. Es gibt viele unterschiedliche bedingte Sprungbefehle, die sich auf unterschiedliche Bedingungen beziehen. Die Bedingung ist im Namen des Befehles angedeutet und bezieht sich entweder direkt auf Flags oder auf einen vorausgegangenen Compare-Befehl (CMP). Beispiele dafür sind:

JC (Jump if Carry, Springe wenn Carryflag gesetzt)
JG (Jump if greater, Springe wenn größer)
JNE (Jump if not equal, Springe wenn nicht gleich)

Die Namen der bedingten Sprungbefehle (JXXX) sind nach folgendem Schema zusammengesetzt:

J : immer erster Buchstabe, "JUMP"
N : evtl. zweiter Buchstabe, "NOT", steht für die Negierung der Bedingung
Z,C,S,O,P : wenn Zero-/Carry-/Sign-/Overflow-/Parityflag gesetzt
E : Equal, gleich
A,B : Above/Below, größer/kleiner bei vorzeichenloser Arithmetik
G,L : Greater/Less, größer/kleiner bei Arithmetik mit Vorzeichen

Daraus ergibt sich eine Fülle von bedingten Sprungbefehlen, die in Tab.7.1 aufgeführt sind. Man sieht dort auch, dass die arithmetischen Bedingungen durch Kombinationen mehrerer Flags ausgewertet werden. Viele Befehle sind mit zwei verschiedenen Mnemonics repräsentiert, z.B. ist JZ (Jump if zero) identisch mit JE (Jump if equal) und erzeugt auch den gleichen Maschinencode. Typisch ist die Kombination eines CMP-Befehles mit einem nachfolgenden bedingten Sprungbefehl (JXXX). Beispiel:

```
    jc ende          ;Jump if Carry nach "ende"
    .
    .                ;wird evtl. übersprungen
    .
ende:
```

Befehl	Sprungbedingung	Sprungbed. dt.	Flags
		Direkte Abfrage von Flags	
JE|JZ	equal|zero	gleich|Null	ZF=1
JNE|JNZ	not equal|zero	ungleich|ungleich Null	ZF=0
JS	signed	Vorzeichen negativ	SF=1
JNS	not signed	Vorzeichen positiv	SF=0
JP|JPE	parity|parity even	gerade Parität	PF=1
JNP|JPO	no parity|parity odd	ungerade Parität	PF=0
JO	overflow	Überlauf	OF=1
JNO	no overflow	kein Überlauf	OF=0
JC	carry	Übertrag	(CF=1)
JNC	no carry	kein Übertrag	(CF=0)
		Arithmetik mit Vorzeichen	
JL|JNGE	less|not greater or equal	kleiner	CF $\neq$ OF
JLE|JNG	less or equal|not greater	kleiner oder gleich	(SF $\neq$ OF) oder (ZF=1)
JNL|JGE	not less|greater or equal	nicht kleiner	(SF=OF)
JG|JNLE	greater|not less or equal	größer	(SF=OF) und (ZF=0)
		Vorzeichenlose Arithmetik	
JA|JNBE	above|not below or equal	oberhalb	(CF=0) und (ZF=0)
JB|JNAE	below|not above or eq.	unterhalb	(CF=1)
JNA|JBE	not above|below or equal	nicht oberhalb	(CF=1) oder (ZF=1)
JNB|JAE	not below|above or equal	nicht unterhalb	(CF=0)

Tabelle 7.1: Bedingte Sprungbefehle

```
Schleifenanfang:
    .
    .
    .
    cmp cx,10
    jb Schleifenanfang   ;Springe nach "Schleifenanfang" wenn cx<10
```

Bedingte Sprungbefehle werden meistens benutzt um Verzweigungen und Schleifen zu realisieren, sie sind von elementarer Wichtigkeit für die Assemblerprogrammierung. Es gibt aber eine wichtige Einschränkung für die bedingten Sprünge:

Alle bedingten Sprünge können nur Ziele im Bereich von -128 Byte bis +127 Byte erreichen. Liegt ein Sprungziel weiter entfernt, wird die Assemblierung mit einer Fehlermeldung abgebrochen. Beispiel:

```
  jz ende   ;Sprungziel ende zu weit entfernt!!!
            ;Fehlermeldung beim Assemblieren
```

In diesem Fall muss man eine Hilfskonstruktion mit einem unbedingten Sprungbefehl benutzen; dieser kann ja beliebige Entfernungen überbrücken.

```
  jnz hilfsmarke
  jmp ende      ;Erreicht weit entferntes Sprungziel
hilfsmarke:
```

7.3 Verzweigungen und Schleifen

Verzweigungen und Wiederholungsschleifen werden in Assemblersprache durch Sprungbefehle realisiert. Eine Verzweigung mit zwei Zweigen wird grundsätzlich folgendermaßen aufgebaut (die Namen der Marken sind natürlich frei wählbar):

```
    cmp Operand1, Operand2
    jxxx Wahr-Zweig     ; Bedingter Sprungbefehl
    .
    .   ;Falsch-Zweig, wird ausgeführt, wenn Bedingung falsch
    .
    jmp Verzweigungsende
Wahr-Zweig:
    .
    .   ;Wahr-Zweig, wird ausgeführt, wenn Bedingung wahr
    .
Verzweigungsende:
```

Der Wahrzweig kann auch entfallen, dann hat man einen bedingt ausgeführten Befehlsblock. Ein Beispiel für obige Konstruktion ist:

```
    mov dx, pixelnr
    cmp dx,[MaxPixelnr]     ;Vergleiche mit MaxPixelNr
    ja Fehler               ;jump if above -> Fehler
                            ;Springe zu Marke Fehler, wenn dx größer
    mov [Fehlerflag],0
    jmp Verzweigungsende
Fehler:
    mov [Fehlerflag],1
Verzweigungsende:
```

Bei den Schleifen muss man zwei Hauptvarianten unterscheiden:

Nicht abweisende Schleifen Die Abbruchbedingung wird erst nach der ersten Ausführung geprüft, die Schleife wird mindestens einmal ausgeführt (C: do - while)

Abweisende Schleifen Die Abbruchbedingung wird schon vor der ersten Ausführung geprüft, möglicherweise wird die Schleife gar nicht ausgeführt (C: while oder for)

Die Abbruchbedingungen der Schleifen können das Erreichen eines bestimmten Zählwertes sein (Zählschleifen) oder datenabhängig formuliert werden. Die Grundkonstruktion der nicht abweisenden Schleife kann folgendermaßen aussehen:

```
    Initialisierung der Schleife
```

```
Schleifenstart:
    Schleifenrumpf (Befehle)
    Schleifenbedingung aktualisieren (z.B. Zählwert ändern) und auswerten,
    bedingter Sprung zu "Schleifenstart"
```

Als Beispiel betrachten wir die Initialisierung eines Wort-Feldes mit Nullworten in einer Zählschleife:

```
    mov si, 0
    mov bx, offset feld
Schleifenstart:
    mov [bx+si], 0  ;0 in Feld schreiben
    add si,2        ;um 2 erhöhen da Wortfeld
    cmp si,256
    jne Schleifenstart
```

Die Grundkonstruktion der abweisenden Schleife sieht wie folgt aus:

```
    Initialisierung der Schleife
Schleifenstart:
    Schleifenbedingung aktualisieren (z.B. Zählwert ändern)und auswerten
    bedingter Sprung zu "Schleifenende"
    Schleifenrumpf (Befehle)
    unbedingter Sprung zu "Schleifenstart"
Schleifenende:
```

In einem Beispiel wird die Ausgabe einer durch Nullbyte begrenzten Zeichenkette (ASCIIZ-String) gezeigt:

```
    mov bx, offset Zeichenkette
Schleifenstart:
    mov dl,[bx]     ; Nächstes Zeichen
    cmp dl,0        ; Begrenzungszeichen
    je Schleifenende
    mov ah,2        ; Schleifenrumpf: Ausgabe
    int 21h         ; mit Int 21h, Funktion 2
    inc bx          ; um 1 erhöhen, da Byte-Feld
    jmp Schleifenstart
Schleifenende:
```

Bei der Programmierung von Schleifen gibt es viele Variationsmöglichkeiten: Die Zähler können Register oder Speichervariable sein, die Zähler können erhöht oder erniedrigt werden, die Bedingungen können arithmetisch sein oder sich direkt auf Flags beziehen u.a.m.

7.4 Die Loop-Befehle

7.4.1 Loop

Der Loop-Befehl ist ein Spezialbefehl für die Programmierung von Schleifen (engl. *Loop* = Schleife). Der Loop-Befehl erniedrigt CX bzw. in 32-Bit-Segmenten ECX und springt anschließend zu einem als Operanden angegebenen Sprungziel, falls CX bzw. ECX nicht 0 ist. Damit lassen sich sehr einfach Zählschleifen programmieren, deren Zählindex in CX/ECX geführt wird und abwärts bis auf 0 läuft. Im folgenden Beispiel wird ein Datenbereich von 1024 Byte mit 0FFh initialisiert:

```
    mov bx, offset Datenbereich
    mov ecx,256
Schleifenstart:
    mov [bx], 0FFFFFFFFh     ; 8 Byte schreiben
    add bx,8
    loop schleifenstart
```

Ein Problem kann sich ergeben, wenn die Anzahl der Schleifendurchgänge, also der Startwert in ECX, variabel gehalten ist. CX/ECX=0 wird ja *vor* der Überprüfung auf 0 dekrementiert. Es ergibt sich also im ersten Durchgang ein Wert von 0FFFFh in CX oder sogar 0FFFFFFFFh in ECX. Die Schleife wird dann ausgeführt bis der Wert zu 0 geworden ist, also (2^{16}) bzw. (2^{32}) mal. Dies ist in der Regel ungewollt. Um das Problem zu vermeiden, gibt es einen speziellen Sprungbefehl: jcxz, (jump if cx zero) und jecxz (jump if ecx zero) wird ausgeführt wenn CX=0 bzw. ECX=0 ist.

7.4.2 Loope/Loopz

Der Befehl Loope (Loop while equal), gleichwertig kann loopz (loop while zero) benutzt werden, macht den Sprung von zwei Bedingungen abhängig:

- CX ungleich 0
- zf=1

Nur wenn beide Bedingungen erfüllt sind, wird der Sprung ausgeführt. Anders ausgedrückt: Die Schleife wird abgebrochen, wenn CX=0 oder zf=0 ist. Loope/Loopz stellt also für die Schleife ein zweites Abbruchkriterium zur Verfügung. Dies setzt voraus, dass innerhalb des Schleifenrumpfes mindestens ein Befehl ausgeführt wird, der das Zeroflag setzt/löscht, z.B. ein bitweise logischer oder arithmetischer Befehl.

7.4.3 Loopne/Loopnz

Der Befehl Loopne (Loop while not equal), gleichwertig kann loopnz (loop while not zero) benutzt werden, macht den Sprung ebenfalls von zwei Bedingungen abhängig:

- CX ungleich 0
- zf=0

Nur wenn beide Bedingungen erfüllt sind, wird der Sprung ausgeführt. Anders ausgedrückt: Die Schleife wird abgebrochen, wenn CX=0 oder zf=1 ist. Wie bei Loope/Loopz muss also im Schleifenrumpfes mindestens ein Befehl stehen, der das Zeroflag setzt/löscht.

7.5 Testfragen

1. Ergänzen Sie in dem folgenden Programmstück die fehlenden Befehle oder Operanden! (???)

```
; Ausgabe der Ziffern von '9' abwärts bis '0'
    mov dl, ???
Schleifenstart:
    mov ah,2        ;DOS-Funktion Zeichenausgabe
    int 21h
    ???
    cmp ???,???
    jae Schleifenstart
```

2. Geben Sie den Inhalt des Registers AX nach dem folgenden Programmstück an!

```
    mov cx,0F0h
    mov ax,0
Schleifenstart:
    inc cx
    dec ax
    cmp cx,100h
    jb Schleifenstart
```

3. Finden Sie die Fehler in dem folgenden Programmstück!

```
; Belegen eines Wortfeldes mit dem Wert 0
.DATA
    Feld DW 20 DUP(?)
.CODE
    mov bx,1
Schleifenstart:
    mov [Feld+bx],0
    inc bx
    cmp bx,20
    je Schleifenstart
```

Lösungen auf Seite 129.

Kapitel 8

Rechnen in Assembler: Arithmetische Befehle

Der i80x86 hat Befehle zur Addition, Subtraktion, Division und Multiplikation mit vorzeichenlosen Zahlen und vorzeichenbehafteten Zahlen (Zahlen im Zweierkomplement). Für die vorzeichenbehafteten Zahlen gibt es ausserdem einen Befehl zur Vorzeichenumkehr. Bei der Addition und Subtraktion werden die gleichen Befehle benutzt, egal ob man mit oder ohne Vorzeichen rechnet. Bei Multiplikation und Division gibt es unterschiedliche Befehlsvarianten für vorzeichenlose und vorzeichenbehaftete Zahlen. Im folgenden Abschnitt sollen nun zuerst diese beiden Zahlenformate erklärt werden.

8.1 Die Darstellung von ganzen Zahlen

Ein Mikroprozessorsystem verarbeitet immer Bitmuster in Einheiten zu 8, 16, 32 oder mehr Bit. Erst durch die Art der Verarbeitung wird diesem Bitmuster eine bestimmte Bedeutung zugewiesen. Wende ich z.B. einen arithmetischen Maschinenbefehl auf ein Bitmuster an, so wird es als Zahl interpretiert, eine Ausgabe auf den Bildschirm interpretiert das gleiche Bitmuster dagegen als darstellbares Zeichen des aktuellen Zeichensatzes. Betrachten wir ein Beispiel: Ein Byte habe den Inhalt 01000011b = 43h = 67d Dies kann interpretiert werden als:

- ASCII-Zeichen 'C'
- Vorzeichenlose oder vorzeichenbehaftete 8-Bit-Zahl: 67d = 43h
- als Maschinenbefehl
- Bitmuster um die Interrupts 0,1 und 6 freizugeben

Wir wollen hier die Interpretation von Bitmustern als Zeichen und ganze Zahlen betrachten.

Für die Ausgabe auf einen Bildschirm oder Drucker muss ein definierter Vorrat an Buchstaben, Ziffern und sonstigen Zeichen verfügbar sein, der *Zeichensatz* . Es sind verschiedene Zeichensätze in Gebrauch, z.B. der *ASCII-Zeichensatz* (American Standard Code for Information

Interchange). Da im ASCII-Zeichensatz jedes Zeichen mit 7 Bit verschlüsselt ist, enthält er 128 Zeichen. Die ersten 32 Zeichen sind Steuerzeichen, wie z.B. Wagenrücklauf, Zeilenvorschub, Tabulator u.a.m. Es gibt auch 8-Bit- und 16-Bit-Zeichensätze. Der Zeichensatz steht in den Geräten hardwaremäßig zur Verfügung, und ordnet jedem Code das Bitmuster des dazu gehörigen Zeichens zu. Soll z.B. das große 'A' des ASCII-Zeichensatzes auf den Bildschirm ausgegeben werden, so muss nur der zugehörige Code 65d an die Grafikkarte übermittelt werden.

Ein Mikroprozessor kann aber ein Bitmuster auch als Zahl interpretieren, dabei wird nach *ganze Zahlen* mit und ohne Vorzeichen sowie *Fließkommazahlen* unterschieden. Um die Darstellung der ganzen Zahlen zu verstehen, betrachten wir zunächst das uns geläufige *Dezimalsystem* , in dem zehn verschiedene Ziffern a_i mit Potenzen der Zahl 10 gewichtet werden. Eine Dezimalzahl mit n Ziffern hat den Wert

$$Z = \sum_{i=0}^{n-1} a_i \cdot 10^i \quad z.B. \quad 123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

In der Mikroprozessortechnik haben die kleinsten Speichereinheiten, die Bits, nur zwei Zustände. Man hat daher nur die Ziffern 0 und 1 zur Verfügung und stellt die Zahlen im Binärsystem dar. Der Wert einer *vorzeichenlosen Binärzahl* (unsigned binary numbers) ist:

$$Z = \sum_{i=0}^{n-1} a_i \cdot 2^i \tag{8.1}$$

Dazu ein Beispiel:

$$\begin{aligned} 11100101b &= 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 128 + 64 + 32 + 4 + 1 = 229d \end{aligned}$$

Um Binärzahlen von Dezimal- u.a. Zahlen zu unterscheiden, wird an die Ziffernfolge der Buchstabe 'b' angehängt (1101b) oder die Zahlenbasis als tiefgestellter Index (1101_2). Da in der Mikroprozessortechnik immer die Bitstellenzahl begrenzt ist, ist auch der Zahlenbereich begrenzt. Hier kann man mit n Bit insgesamt die Zahlen von 0 bis $2^n - 1$ darstellen, bei 8 Bit z.B. 0 bis 255. Zahlen ausserhalb dieses Bereichs sind nicht darstellbar und eine Operation, deren Ergebnis über eine der Grenzen hinaus führt, ergibt ein falsches Ergebnis. Diese Bereichsüberschreitung wird vom Mikroprozessor mit dem *Übertragsflag* (Carryflag) angezeigt. Dagegen wird bei einem Überlauf (s.u.) auf das höchstwertige Bit zwar das Überlaufsflag gesetzt, dies bedeutet hier aber keinen Fehler; ein Beispiel dafür ist 127+1=128 bei 8-Bit-Zahlen.

Bei einer Bereichsüberschreitung landet man also nach der größten darstellbaren Zahl wieder bei Null bzw. umgekehrt. Das erinnert an einen Ring oder eine Uhr und man kann tatsächlich den Zahlenbereich bei ganzen Zahlen sehr anschaulich durch den sog. Zahlenring darstellen.

Um einen Zahlenbereich zu erhalten, der auch negative Zahlen erlaubt, werden die Zahlen im *Zweierkomplement* (signed binary numbers) dargestellt. Dabei gibt es nur einen Unterschied zu den vorzeichenlosen Binärzahlen: Die höchstwertige Ziffer wird negativ gewichtet, also:

$$Z = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i \tag{8.2}$$

Bitmuster	Wert dezimal
11111101	253
11111110	254
11111111	255
00000000	0
00000001	1
00000010	2
00000011	3

Tabelle 8.1: Der Übertrag bei Binärzahlen am Beispiel der 8-Bit-Zahlen. Bsp.: Die Operation 255+1 führt zu dem falschen Ergebnis 0, der Fehler wird durch das Übertragsflag (Carry) angezeigt.

Auch dazu ein Beispiel:

$$\begin{aligned} 11100101b &= -1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= -128 + 64 + 32 + 4 + 1 = -27d \end{aligned}$$

Natürlich ist auch hier der darstellbare Zahlenbereich begrenzt und zwar auf $-2^{n-1} \ldots +2^{n-1}-1$. 8-Bit-Zahlen im Zweierkomplement können beispielsweise den Zahlenbereich $-128 \ldots +127$ darstellen. Das höchstwertige Bit ist bei negativen Zahlen gesetzt (=1) und bei nichnegativen zahlen nicht gesetzt (=0), man bezeichnet es daher auch als *Vorzeichenbit* . Die Zweierkomplementzahlen haben die angenehme Eigenschaft, dass die positiven Zahlen nahtlos an die negativen Zahlen anschließen. Nehmen wir als Beispiel wieder die 8-Bit-Zahlen im Zweierkomplement und betrachten folgenden Ausschnitt aus dem Zahlenbereich:

Bitmuster	Wert dezimal
11111101	-3
11111110	-2
11111111	-1
00000000	0
00000001	1
00000010	2
00000011	3

Tabelle 8.2: Der Anschluss der positiven an die negativen Zahlen im Zweierkomplement am Beispiel der 8-Bit-Zahlen. -1 + 1 führt zu dem richtigen Ergebnis 0.

Man sieht, dass man mit den ganz normalen Additions- und Subtraktionsbefehlen problemlos von den positiven zu den negativen Zahlen gelangen kann, wenn man das Übertragsflag (Carry) ignoriert, und genau so arbeitet ein Mikroprozessor! Bei der Arbeit mit den Zweierkomplementzahlen lauert allerdings eine andere Gefahr: Ein Übertrag auf das Vorzeichenbit, der sog. *Überlauf* ändert nämlich das Vorzeichen der Zahl! Dies passiert allerdings nur, wenn nicht gleichzeitig auch ein Übertrag entsteht und Mikroprozessoren setzen auch nur dann das Überlaufflag. Betrachten wir wieder einen Ausschnitt aus dem Zahlenbereich der 8-Bit-Zahlen im Zweierkomplement:

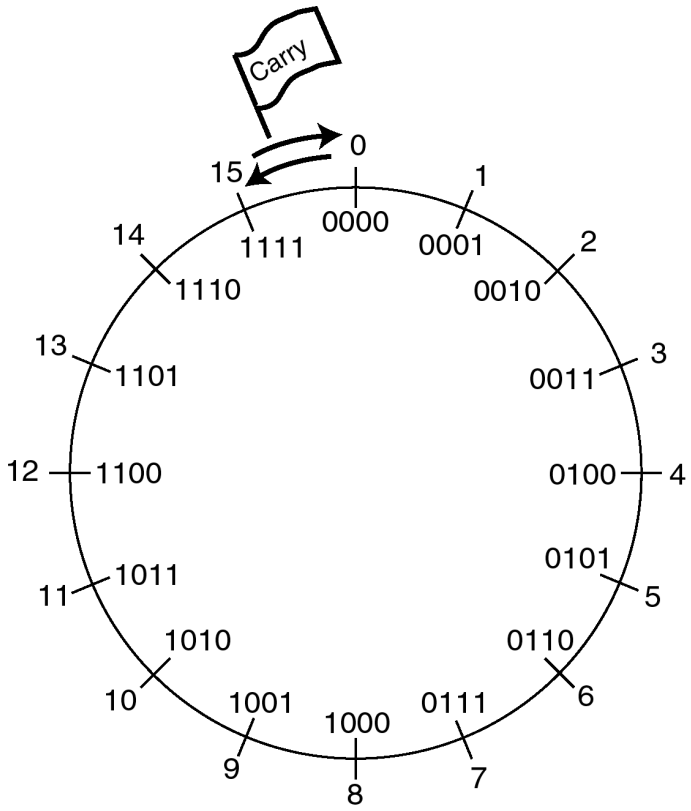


Abbildung 8.1: Der Zahlenring für die vorzeichenlosen 4-Bit-Zahlen. Die Bereichsüberschreitung wird durch das Übertragsflag (Carry) angezeigt.

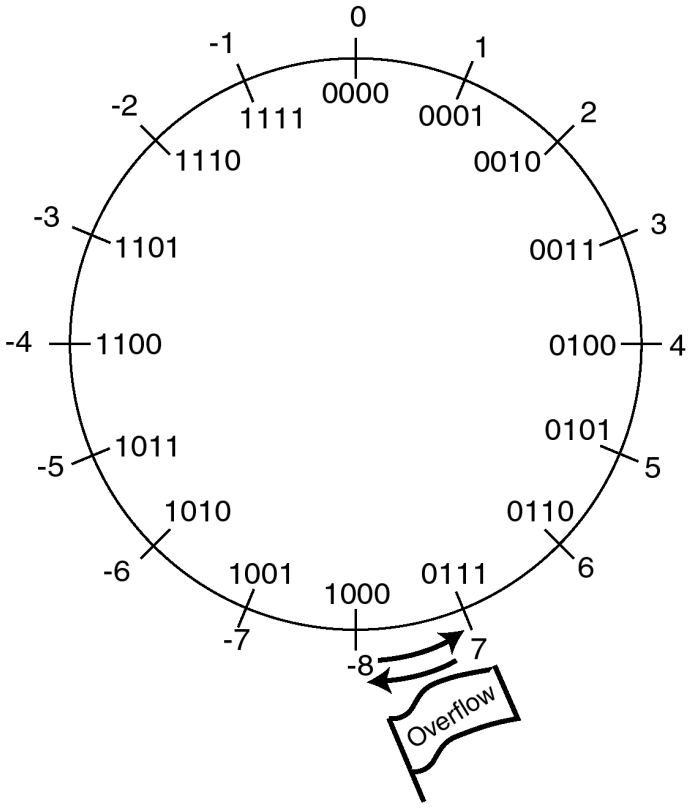


Abbildung 8.2: Der Zahlenring für die 4-Bit-Zahlen im Zweierkomplement. Die Bereichsüberschreitung wird durch das Überlaufsflag (Overflow) angezeigt.

Bitmuster	Wert dezimal
01111101	125
01111110	126
01111111	127
10000000	-128
10000001	-127
10000010	-126
10000011	-125

Tabelle 8.3: Die „Bruchstelle“ zwischen positiven an die negativen Zahlen beim Überlauf am Beispiel der 8-Bit-Zahlen im Zweierkomplement. Bsp.: Die Operation 127+1 führt zu dem falschen Ergebnis -128, der Fehler wird durch das Überlaufsflag (Overflow) angezeigt.

Auch die Zweierkomplement-Zahlen können sehr schön im Zahlenring dargestellt werden, die Bereichsüberschreitung wird hier durch das Überlaufflag angezeigt. Die Vorzeichenumkehr einer Zahl im Zweierkomplement wird bewirkt durch *Invertieren aller Bits und anschließendes Inkrementieren*. Dies kann leicht gezeigt werden, wenn man von Gl.8.2 ausgeht. Es ist $\bar{Z}$ das bitweise invertiertes Z und $(1 - a_i) = \bar{a}_i$

$$\begin{aligned}
\bar{Z} &= -(1 - a_{n-1}) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (1 - a_i) \cdot 2^i \\
&= -2^{n-1} + a_{n-1} \cdot 2^{n-1} + (2^{n-1} - 1) - \sum_{i=0}^{n-2} a_i \cdot 2^i \\
&= a_{n-1} \cdot 2^{n-1} - \sum_{i=0}^{n-2} a_i \cdot 2^i - 1 = -Z - 1
\end{aligned}$$

Damit ergibt sich also für die vorzeichenumgekehrte Zahl $-Z$

$$-Z = \bar{Z} + 1 \tag{8.3}$$

Als Beispiel betrachten wir die Vorzeichenumkehr von -11d = 11110101b; nach Invertieren ergibt sich 00001010b und nach dem anschließenden Inkrementieren 00001011b = 11d.

8.2 Addition und Subtraktion

Die einfachsten arithmetischen Befehle sind INC (Increment) und DEC (Decrement) für die Erhöhung und Erniedrigung um eins. INC und DEC setzen die fünf Flags OF, SF, ZF, AF und PF. Im Unterschied zu ADD und SUB wird also CF nicht gesetzt. Beispiele:

```
inc cx   ; erhöht cx um 1
dec [Feld+bx]   ;erniedrigt eine Speichervariable um 1
```

Mit dem Additionsbefehl ADD kann man zwei Operanden gleicher bitbreite addieren, das Additionsergebnis wird im ersten Operanden abgelegt. Die Operanden können jeweils 8-, 16- oder 32-Bit-Zahlen in Registern, Speichervariablen oder Direktwerten sein. (Nicht zwei Speichervariable!) Der Befehl ADD setzt sechs Flags: OF, SF, ZF, AF, CF und PF. Beispiele:

```
add ax,bx   ; addiert ax und bx, Summe nach ax
add cl,6    ; addiert cl und 6, Summe nach cl
add [zaehler],ax    ; addiert ax zu zaehler, Ergebnis in zaehler
add ebx,[dwort]    ; addiert dwort zu ebx, Ergebnis in ebx
```

Der Befehl ADC (Add with Carry) addiert zusätzlich als dritten Operanden das Carryflag mit der Wertigkeit des LSB zum Ergebnis. Damit können Ganzzahlwerte beliebiger Bitbreite addiert werden. Man addiert zunächst die niederwertigste Einheit (Wort/Doppelwort) mit ADD und addiert die höherwertigen Einheiten mit ADC, so dass ein durch gesetztes Carryflag angezeigter Übertrag bei der darauffolgenden Addition einbezogen wird. Beispiel:

```
; Addition zweier 64-Bit-zahlen
    add eax,[Zahllow]   ; addiert niederwertige Doppelworte
    adc edx,[Zahlhigh]  ; addiert höherwertige Doppelworte
```

Für die Subtraktion gibt es die Befehle SUB (Subtract) und SBB (Subtract with Borrow). Diese sind formal genauso aufgebaut wie ADD/ADC und arbeiten auch genauso zusammen. Zu beachten ist, *dass bei der Subtraktion der zweite Operand vom ersten subtrahiert wird.*

```
sub cx,bx   ; subtrahiert bx von cx, Ergebnis nach cx
sub al,6    ; subtrahiert 6 von al, Erg. nach al
sub [zaehler],ax    ; subtrahiert ax von zaehler, Ergebnis in zaehler
sub esi,[dwort]    ; subtrahiert dwort von esi, Ergebnis in ebx
```

Der Befehl CMP, Compare, ist eigentlich eine Sonderform von SUB. Er arbeitet intern genau wie SUB, schreibt allerdings das Ergebnis nicht zurück in den ersten Operanden. Der Sinn von CMP liegt im Setzen der Flags.

8.3 Multiplikation

8.3.1 Vorzeichenlose Multiplikation: MUL

Der Befehl erlaubt die Multiplikation zweier Operanden gleicher Bitbreite und berücksichtigt, dass das Ergebnis doppelt so viele Bit haben kann. Man unterscheidet die Byte, Wort- und Doppelwortmultiplikation.

Bezeichnung	Multiplikator	Multiplikand		Ergebnis
Bytemultiplikation	8 Bit	8 Bit	→	16 Bit
Wortmultiplikation	16 Bit	16 Bit	→	32 Bit
Doppelwortmultiplikation	32 Bit	32 Bit	→	64 Bit

Byte-Multiplikation Syntax: `MUL` *reg8/mem8*
Der genannte Operand wird mit Register AL multipliziert, das Ergebnis hat 16 Bit und wird in AX abgelegt. Beispiel:
```
mul bl ; multipliziert bl mit al, Ergebnis in ax
```

Wort-Multiplikation Syntax: `MUL` *reg16/mem16*
Der genannte Operand wird mit Register AX multipliziert, das Ergebnis umfasst 32 Bit und wird in DX-AX abgelegt, welche dabei zu einem 32-Bit-Register zusammengeschaltet werden (High Word in DX). Beispiel:
```
mul word ptr [di] ; multipliziert 16-Bit-Speichervar. mit ax
```

Doppelwort-Multiplikation (ab 386) Syntax: `MUL` *reg32/mem32*
Der genannte Operand wird mit Register EAX multipliziert, das Ergebnis umfasst 32 Bit und wird in EDX-EAX abgelegt.Beispiel:
```
mul ecx ; multipliziert ecx mit eax, Ergebnis in edx-eax
```

Der MUL-Befehl setzt zwei FLags: Es gilt OF=CF=0, wenn die obere Hälfte der Ergebnisbits Null ist, CF=OF=1 sonst. Auf Grund der doppelt so breiten Ergebnisregister kann ein Fehler nicht passieren.

8.3.2 Vorzeichenbehaftete Multiplikation: IMUL

Der IMUL-Befehl war beim i8086 formal und syntaktisch das genaue Gegenstück zum MUL-Befehl, d.h. er arbeitete mit einem Operanden wie im vorigen Abschnitt beschrieben. Später (286, 386) wurden für den IMUL-Befehl viele Varianten ergänzt, so dass man ihn heute mit einem, zwei oder drei Operanden benutzen kann.

IMUL mit einem Operanden Diese Variante entspricht genau dem MUL-Befehl, kann also zur Byte-, Wort und Doppelwort-Multiplikation benutzt werden und verwendet Operanden und Register in der gleichen Weise. (Der Operand (Multiplikator) kann ein Register oder eine Speichervariable mit 8, 16 oder 32 Bit sein. Multiplikand ist AL, AX oder EAX, Ergebnis in AX, DX-AX oder EDX-EAX.)

IMUL mit zwei Operanden Die beiden Operanden sind Multiplikand und Multiplikator, das Ergebnis wird im ersten Operanden abgelegt. (wie z.B. bei ADD)

Syntax: `IMUL` *reg16, Direktwert8/16*
`IMUL` *reg32, Direktwert8/32*
`IMUL` *reg16, reg16/mem16*
`IMUL` *reg32, reg32/mem32*

Beispiel: `imul ebx,ecx ; Multipliziert ebx mit ecx, Erg. nach ebx`

IMUL mit drei Operanden Der erste Operand ist das Zielregister, der zweite und dritte sind sind Multiplikand und Multiplikator. Der Multiplikator muss hier ein Direktwert sein.

Syntax: `IMUL` *reg16, reg16/mem16, Direktwert8*
`IMUL` *reg16, reg16/mem16, Direktwert16*
`IMUL` *reg32, reg32/mem32, Direktwert8*
`IMUL` *reg32, reg32/mem32, Direktwert32*

Beispiel: `imul edi,ebx,5 ; Multipliziert ebx mit 5, Erg. nach edi`

Der IMUL-Befehl setzt die beiden Flags CF und OF. Die IMUL-Variante mit einem Operanden setzt die Flags wie der MUL-Befehl: OF=CF=0, wenn die obere Hälfte der Ergebnisbits Null ist, CF=OF=1 sonst. Die anderen IMUL-Varianten setzen CF=OF=1, wenn das Ergebnisregister zu klein ist für das Ergebnis, sonst OF=CF=0. Hier zeigen die gesetzten Flags also einen ernsthaften Fehler an!

Ein gute Empfehlung für die Praxis sind die Varianten von IMUL mit zwei Operanden. Man kann positive und negative Zahlen bis ca. 2 Milliarden berechnen und die Syntax ist ähnlich zu vielen gewohnten Befehlen. Beispiele:

```
;Berechnung von 655*76
    mov cx,76
    mov bx,655
    imul bx,cx
    jc Fehler   ; reichen 16 Bit für das Ergebnis?

;Berechnung von 2000h * 32-Bit-Speichervariable
    mov eax,2000h
    imul eax,[Var32]
    jc Fehler   ; reichen 32 Bit für das Ergebnis?
```

8.4 Division

Der Divisionsbefehl arbeitet mit festen Registern: Als Operand wird nur der Divisor (Teiler) genannt, Dividend (Das Geteilte) und Ergebnisregister sind fest, also implizite Operanden. *Dabei wird immer vorausgesetzt, dass das Ergebnis nur halb so viele Bits umfasst, wie der Dividend!* Das Ergebnisregister hat daher nur halb so viele Bit wie der Dividend. Ergibt die Rechnung ein zu großes Ergebnis, so ereignet sich ein *Divisionsfehler* , der den Prozessor in einen sog. Ausnahmezustand versetzt. Der dadurch aufgerufene Exception-Handler beendet in der Regel das Programm! Divisionsfehler können leicht passieren, wenn durch kleine Zahlen geteilt wird. Division durch Null führt immer zum Divisionsfehler.

Bei der Division entstehen zwei Resultate: Ein ganzzahliges *Divisionsergebnis* und ein ganzzahliger *Divisionsrest* .

Für die Division vorzeichenloser Zahlen wird DIV benutzt, für die Division vorzeichenbehafteter Zahlen IDIV. Man unterscheidet Byte- Wort- und Doppelwortdivision, der Operand(Divisor) bestimmt die Art der Division.

Bezeichnung	Dividend	Divisor		Ergebnis	Rest
Bytedivision	16 Bit (AX)	8 Bit (Operand)	→	8 Bit (AL)	8 Bit (AH)
Wortdivision	32 Bit (DX-AX)	16 Bit (Operand)	→	16 Bit (AX)	16 Bit (DX)
Doppelwortdivision	64 Bit (EDX-EAX)	32 Bit (Operand)	→	32 Bit (EAX)	32 Bit (EDX)

Byte-Division Syntax: `DIV/IDIV` *reg8/mem8*

Der Inhalt des Reg. AX wird durch den Operanden geteilt, das Ergebnis wird in AL abgelegt, der Rest in AH. Beispiel:

```
div bl ; dividiert ax durch bl, Ergenis in al, Rest in ah
```

Wort-Division Syntax: `DIV/IDIV` *reg16/mem16*

Dividiert den Inhalt von DX-AX durch den Operanden, Ergebnis wird in AX abgelegt, Rest in DX. Beispiel:

```
idiv word ptr [di] ;dividiert dx-ax durch 16-Bit-Speichervar.
;Ergebnis in ax, Rest in dx, vorzeichenrichtig
```

Doppelwort-Division Syntax: `DIV/IDIV` *reg32/mem32*

Dividiert den Inhalt von EDX-EAX durch den Operanden, Ergebnis wird in EAX abgelegt, Rest in EDX. Beispiel:

```
idiv ebx ;dividiert edx-eax durch ebx
;Ergebnis in eax, Rest in edx, ohne Vorzeichen
```

In den folgenden Beispielen soll noch einmal die Entstehung von Divisionsfehlern verdeutlicht werden:

```
; 1.Beispiel, zu berechnen 823/4
    mov ax,823
    mov bl,4
    div bl      ; Byte-Division da bl
    ; Ergebnisse:
    ;   al=205  (Divisionsergebnis)
    ;   ah=3    (Divisionsrest)


; 2.Beispiel, zu berechnen 823/2
    mov ax,823
    mov bl,2
    div bl      ; Byte-Division da bl
    ; Ergebnisse:
    ;   Keine, da Divisionsfehler und Programmabbruch!!!
    ;   Grund: Ergebnis 410 ist zu groß für Register al


; 3.Beispiel, zu berechnen 823/2
    mov ax,823
    mov bx,2
```

```
    div bx      ; Wort-Division da bx
    ; Ergebnisse:
    ;   meist Divisionsfehler und Programmabbruch!!!
    ;   Grund:
    ;   Bei der Wortdivision wird DX-AX durch den Divisor (hier BX) geteilt
    ;   DX wurde aber nicht auf einen definierten Inhalt gesetzt.
    ;   sobald DX größer als 1 ist, ist das Divisionsergebnis größer als FFFFh
    ;   und Register AX ist zu klein.
    ;   bei DX=1 erhalten wir ein unerwartetes Ergebnis


; 4.Beispiel, zu berechnen 823/2
    mov ax,823
    mov bx,2
    mov dx,0
    div bx      ; Wort-Division da bx
    ; Ergebnisse:
    ;   ax=411 (Divisionsergebnis)
    ;   dx=0   (Divisionsrest)
```

Ein Divisionfehler kann nur sicher vermieden werden, wenn

a) Die höherwertige Hälfte des Dividenden kleiner als der Divisor ist,
b) Abgefragt wird, ob der Divisor ungleich Null ist.

8.5 Vorzeichenumkehr: NEG

Mit NEG, Negate, kann das Vorzeichen einer vorzeichenbehafteten Zahl umgekehrt werden. NEG hat einen Operanden, dieser kann eine Register oder Speichervariable sein mit 8, 16 oder 32 Bit. Wie eine Vorzeichenumkehr auf Bitebene ausgeführt wird, ist in Abschnitt 8.1 beschrieben. Ein Beispiel für die Verwendung von NEG ist die Bildung des Absolutbetrages im folgenden Codeabschnitt:

```
;Bildung des Absolutbetrages von eax
    cmp eax,0
    jge fertig
    neg eax
fertig:
```

8.6 Beispiel

In einem abschließenden Beispiel soll ein arithmetischer Ausdruck in Assembler berechnet werden:

```
; Berechnung von
;
;        2*A + B*C
;   X = ------------
;           D-E
;
; Alle Variablen haben 32 Bit und es wird vorausgesetzt,
; dass das Ergebnis ebenfalls mit 32-Bit darstellbar ist

    mov eax,[A]
    imul eax,2      ;eax = 2*A
    mov ebx,[B]
    imul ebx,[C]    ;ebx = B*C
    add eax,ebx     ;eax = 2*A + B*C
    mov ecx,[D]
    sub ecx,[E]     ;ecx = D-E
    mov edx,0       ;Vorbereitung Division
    idiv ecx        ;eax=(2*A + B*C) / (D-E)
    mov [X],eax     ;zuweisen an x
    ; Rest in edx, Verwendung beliebig
```

8.7 Testfragen

1. ```
   add ax
   adc bx,ax,cf
   mul eax,ebx
   mul 80h
   imul ax,bx,cx
   div edx,eax
   ```

   Finden Sie die syntaktisch fehlerhaften Befehle!

2. Bestimmen Sie, welchen Inhalt die Register AX, BX, CX, DX und DI nach der Ausführung der folgenden Befehle haben!

   ```
   mov cx,20h
   sub cx,90h
   mov dx,90h
   add dx,cx
   mov ax,50h
   mov di,100h
   div di
   imul bx,di,3
   ```

3. Bestimmen Sie, welchen Inhalt die Register AX, und CX nach der Ausführung der folgenden Befehle haben!
   ```

```
mov cx,20h
imul cx,2
imul ax,cx,2
imul cx,ax,2
```

4. Bestimmen Sie, welchen Inhalt die Register AX, BX und DX nach der Ausführung der folgenden Befehle haben!

```
mov dx,50h
mov ax,5h
mov bx,100h
div bx
div bx
```

5.
```
;Berechnung von 123456h / 11h
mov eax,123456h
mov ebx,11h
div ebx
```

   Prüfen Sie den obigen Code, kann es Probleme geben?

Antworten auf Seite 129.

Kapitel 9

Stack und Stackbefehle

9.1 Stackorganisation

Ein Stack, zu deutsch *Stapel* , ist ein Abschnitt im Hauptspeicher, der auf ganz spezielle Art verwaltet wird, nämlich als Last in – First out. Dies bedeutet, das zuletzt eingespeicherte Element wird als erstes wieder herausgegeben. Man kann dies mit einem Stapel Teller vergleichen: Der zuletzt aufgelegte Teller ist der, der auch als erster wieder herausgenommen wird. Mit guter Kenntnis der Verhältnisse kann man allerdings auch ein Element aus der Mitte des Stack auslesen (ähnlich wie man mit etwas Geschick auch einen Teller mitten aus dem Stapel ziehen kann). Das Konzept des Stack ist so elementar, dass es von unserem Prozessor sogar hardwaremäßig unterstützt wird: Es gibt zwei eigens dafür reservierte Register, Stacksegment (SS) und Stackpointer (SP) Für den Stack gilt:

- Auf dem Stack gibt es nur Wortzugriffe!
- Die Register SS und SP, enthalten ständig einen FAR Pointer auf das zuletzt eingespeicherte Wort, den Top of Stack (TOS)
- Der Stack wächst abwärts, also zu kleiner werdenden Adressen hin
- Der Stack wird direkt benutzt durch die Befehle PUSH und POP
- Der Stack wird indirekt benutzt durch die Befehle CALL und RET
- Der Stack kann mit indirekter Adressierung adressiert werden, wenn man BP als Basisregister benutzt.

Der Befehl PUSH speichert ein Wort auf dem Stack. Dabei wird SP um zwei vermindert. Der Befehl POP liest ein Wort vom Stack, dabei wird SP um zwei vergrößert. Der Bereich oberhalb von TOS bis einschließlich TOS ist der Dateninhalt und geschützte Bereich des Stack. Der Bereich unterhalb TOS ist ungültig, d.h. darf frei überschrieben werden (s.Abb.9.1).

Mit den Befehlen PUSH und POP läßt sich der Stack einfach als Last-In-First-Out ansprechen. Dabei wird die automatische Stackverwaltung durch die Register SS und SP benutzt.

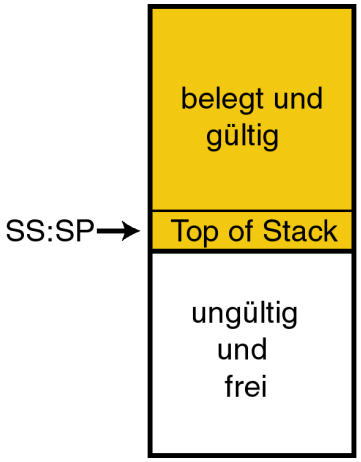


Abbildung 9.1: Aufbau des Stacks. Ein PUSH-Befehl speichert ein neues Wort und verschiebt damit den TOS nach unten.

PUSH Der PUSH-Befehl speichert ein Wort auf dem Stack ab. Dieses wird unterhalb des bisherigen TOS gespeichert und bildet den neuen TOS. SP wird dabei um zwei vergrößert, der Stack ist also um zwei Byte zu den niedrigen Adressen hin gewachsen.

POP Der POP-Befehl liest das TOS-Wort vom Stack und vergrößert SP um zwei. Der Stack ist also zwei Byte kleiner geworden, es gibt ein neues TOS.

PUSH und POP haben je einen Operanden, der ein 16-Bit-Register oder ein 16-Bit-Speicherplatz sein kann (ab 80286 auch ein Direktwert) Wichtig: *Am Ende eines Programmes oder Unterprogrammes muss der Stackpointer wieder den gleichen Wert haben wie am Anfang. Deshalb muss die Anzahl der ausgeführten PUSH- und POP-Befehle (oder gleichwertiger Ersatzkonstruktionen) gleich sein!* (stack balancing)

9.2 Stacküberlauf

Wenn durch mehrere PUSH-Befehle das SP-Register den Wert 0 erreicht hat bewirkt ein weiterer PUSH-Befehl einen Stacküberlauf. Dabei werden fehlerhafterweise daten überschrieben, entweder im Stacksegment oder in einem anderen Segment.

9.3 Anwendungsbeispiele

Eine typische Anwendung ist das temporäre Zwischenspeichern von Daten. Beachten Sie im folgenden Beispiel die Reihenfolge der PUSH und POP-Befehle:

```
push dx     ;dx zwischenspeichern
push ax     ;ax zwischenspeichern
mov dx, offset Meldung  ;dx wird gebraucht
mov ah,09
```

```
int 21h
pop ax      ;ax erhält wieder den alten Wert
pop dx      ;dx erhält wieder den alten Wert
```

Manchmal kann auch ein Transport von Daten gut über den Stack abgewickelt werden. Im folgenden Beispiel soll der Inhalt von DS nach ES kopiert werden. Dies kann geschehen durch

```
mov ax,ds   ; DS kann nicht direkt nach ES kopiert werden
mov es,ax   ; AX ist verändert
```

Das gleiche kann auch über den Stack als Zwischenstation geschehen:

```
push ds     ; Keine anderen Register werden verändert
pop es
```

Die Benutzung des Stack erlaubt z.B. auch beliebig tief geschachtelte Schleifen, die alle mit dem Register CX zählen.

9.4 Testfragen

1. ```
   push di
   push dl
   push fs
   push edi
   ```
   Welche Befehle sind fehlerhaft?

2. ```
   push 8
   push 9
   push 10
   pop ax
   pop bx
   pop cx
   ```
   Welchen Inhalt haben die Register ax,bx,cx?

3. ```
       mov cx,10
   mult: imul ax,2
       push ax
       dec cx
       cmp cx,0
       jne mult

       mov di,0
   vom_stack_holen:  pop ax
       mov [Feld+di],ax
   ```
   ```

```
add di,2
cmp di,20
jbe vom_stack_holen
```

Kommentieren Sie diesen Programmabschnitt!

Lösungen auf Seite 130

Kapitel 10

Unterprogramme

Unterprogramme, engl. *Procedures* oder *Subroutines* , sind notwendig, um gute Assemblerprogramme zu schreiben. Die Gründe dafür sind:

- Die Ablaufsteuerung für Teilaufgaben ist zentral und nur einmal vorhanden
- Gute Unterprogramme sind modular und unterstützen die Wiederverwendung
- Eine unnötige Aufblähung des Maschinencodes wird vermieden
- Die Übersicht wird verbessert
- Unterprogramme stellen gute Schnittstellen zu Hochsprachen dar

Mit dem Aufruf des Unterprogramms (CALL) verzweigt der Programmablauf ins Unterprogramm. Das Unterprogramm endet mit dem Return-Befehl (RET). Dieser bewirkt, dass die Ausführung mit dem nächsten Befehl, der auf CALL folgt, fortgesetzt wird. Bei den meisten Unterprogrammen werden Informationen mit dem rufenden Programm ausgetauscht. Das Unterprogramm wird durch *Parameter* gesteuert und liefert Ergebnisse an das rufende Programm zurück. Unterprogramme können ihrerseits wieder Unterprogramme aufrufen. Da Unterprogramme mit den gleichen Register arbeiten müssen, wie das rufende Programm, können nach dem Unterprogramm Register verändert sein. Im folgenden Beispiel bildet ein Unterprogramm den Mittelwert aus AX und BX und gibt ihn in AX zurück.

```
.model small
.stack 100h
.code
start:
mov ax,15       ; Vorbereitung des Unterprogrammaufrufs
mov bx,19       ; Übergabe der Parameter in AX und BX
CALL Mittelwert_ax_bx   ; Unterprogrammaufruf
mov ah,4ch      ; Programmende
int 21h         ;
```

```
PROC Mittelwert_ax_bx
    add ax,bx    ; Summe aus ax und bx nach ax
    shr ax,1     ; durch zwei teilen
    ret          ; Return: Rücksprung
ENDP Mittelwert_ax_bx

    END Start
```

Die Verwendung von PROC und ENDP ist nicht notwendig, ist aber sehr zu empfehlen. Für die Übergabe von Parametern und Ergebnissen gibt es verschiedene Möglichkeiten:

Übergabe in Registern Einfach und gut, üblich in reinen Assemblerprogrammen, z.B. BIOS, DOS. Möglich ist auch die Rückgabe von Ergebnissen in Flags. Nachteil: Anzahl und Umfang der Parameter begrenzt. Bei großen Datenstrukturen muss man diese in Pufferspeichern halten und Zeiger auf diese Daten in Registern übergeben.

Stackübergabe In Hochsprachen implementiert, etwas komplizierter ermöglicht aber (fast) unbegrenzte Übergabe

Über globale Variable führt zu Abhängigkeiten von Variablendefinitionen im rufenden Programm. Rückgabe der Ergebnisse = Seiteneffekt. Schlecht und nur eine Notlösung!

Der Rücksprung an die richtige Stelle im rufenden Programm wird auf folgende Art gesichert:

CALL speichert die Adresse des nächsten Befehles nach CALL, die Rücksprungadresse, im rufenden Programmstück auf den Stack.

RET holt die Rücksprungadresse vom Stack und läd sie in das Instruction Pointer Register (IP), so dass dort fortgesetzt wird.

[1]

Ein wichtiges Thema ist das Verändern von Registern durch Unterprogramme. Es gibt verschiedene Möglichkeiten damit umzugehen:

Unterprogramm ändert keine Register Die sichere Methode: Alle im Unterprogramm benutzten Register werden zu Beginn des Unterprogramms auf den Stack gerettet und vor dem Return-Befehl wieder hergestellt. Dabei werden allerdings auch die Register gesichert, die im rufenden Programm gar nicht mehr gebraucht werden. Dies verlangsamt das Programm. Register für die Ergebnisrückgabe müssen ausgenommen werden.

Unterprogramm kümmert sich nicht um veränderte Register Die intelligente Methode: Der Programmierer muss sich im rufenden Programm um die Rettung von Registerinhalten kümmern, allerdings nur, wenn diese Register noch gebraucht werden. Erfordert Aufmerksamkeit, führt aber zu den schnellstmöglichen Programmen. Die richtige Wahl bei zeitkritischen Programmen.

[1] CALL und RET sind also spezielle Sprungbefehle. In einer 16-Bit-Umgebung codiert der Assembler sie je nach Speichermodell als NEAR oder FAR.

Registergruppen Dies ist ein Kompromiss aus den beiden ersten Lösungen: Ein Teil der Register darf frei im Unterprogramm verändert werden, der Rest muss gesichert und wiederhergestellt werden. In Hochsprachen verwendet.

Tips zum guten Stil

Ein gutes Unterprogramm...

- erledigt eine Aufgabe und nur eine
- ist so kurz wie möglich und so lang wie nötig (z.B. max. 100 Zeilen)
- beginnt mit Kommentaren: Aufgabe des Unterprogrammes, Übergabe der Parameter, veränderte Register, Anmerkungen des Bearbeiters
- kann alleinstehend übersetzt und verstanden werden
- hat einen treffenden Namen

Kapitel 11

Die Gleitkommaeinheit

Die Gleitkommaeinheit, engl. floating point unit, FPU, ist eine relativ eigenständige Verarbeitungseinheit des Prozessors. Sie verfügt über einen eigenen Befehlssatz und eigene Register und sie kann parallel zum Hauptprozessor arbeiten. Die SSE-Verarbeitungseinheiten (ab Pentium III) können mehrere Gleitkommazahlen parallel verarbeiten.

11.1 Gleitkommazahlen

Die Gleitkommaeinheit unterstützt drei Gleitkommaformate, die in Abb.11.1 dargestellt sind. Intern wird immer im 80-Bit-Format gerechnet, die anderen Formate werden beim Laden in den Gleitkommastack in dieses Format umgewandelt. Alle Gleitkommaformate bestehen aus den Anteilen Vorzeichen, Exponent und Mantisse.

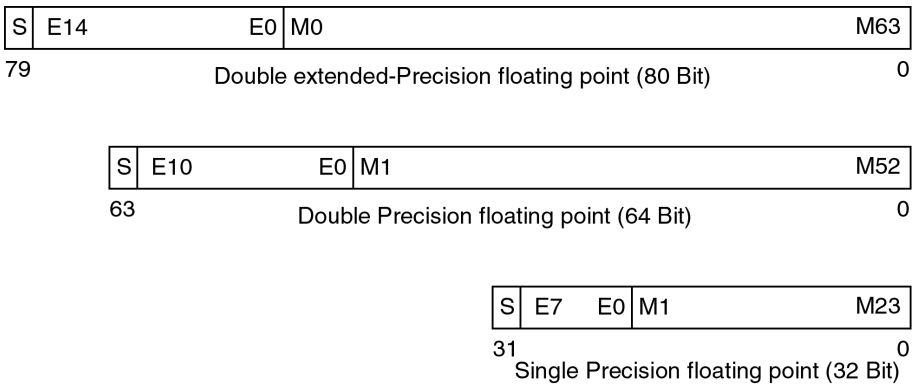


Abbildung 11.1: Die unterstützten Gleitkommaformate

11.2 Aufbau der Gleitkommaeinheit

11.2.1 Die Register der Gleitkommaeinheit

Es gibt 8 *Datenregister* R1 bis R8 zu je 80 Bit in denen die zu verarbeitenden Zahlen liegen und in denen auch die Ergebnisse abgelegt werden. Die Zahlen liegen im Double extended-Precision

floating point-Format (auch Temporary Real-Format genannt) vor (Abb. 11.1).

Das *Statuswort* ist ein 16-Bit-Register, das Flags und Bitfelder mit Informationen über den momentanen Zustand der FPU enthält, dazu gehören: Flags für Genauigkeitsverlust, Unterlauf, Überlauf, Division durch Null, ungültige Operation, Busy, Condition-Flags C1 – C4 und der 3-Bit-Zeiger Top of Stack.

Der wichtigste Bestandteil der Gleitkommaeinheit ist aus der Sicht des Programmierers der Block mit den acht Gleitkommaregistern, die als Stack organisiert sind. Alle arithmetischen Befehle und Transferbefehle beziehen sich auf Register des Stacks.

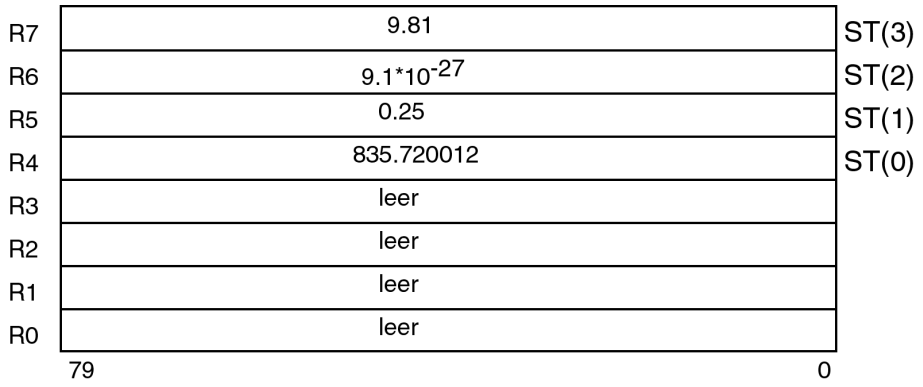


Abbildung 11.2: Registerstack der Gleitkommaeinheit. Zuletzt wurde der Wert 835.710012 geladen.

Im Registerstack werden die Register entsprechend Ihrer Lage zum Top-of-Stack bezeichnet. Die zuletzt geladene Zahl ist ST(0), die davor geladene ST(1) usw.

11.3 Befehlssatz

11.3.1 Datentransportbefehle

Daten können mit den *Ladebefehlen* aus dem Speicher in den Registerstack geladen werden. Beim Laden findet die automatische Typkonvertierung in das intern benutzte 80-Bit-Format statt. Die geladene Zahl ist automatisch das neue ST(0), aus ST(0) wird ST(1) usw. Jedes Laden enthält also implizit ein PUSH. Es gibt mehrere Ladebefehle für verschiedene Datentypen, in den Befehlsnamen steht immer LD für Load:

FLD fp-Variable	Gleitkommazahl mit 32, 64 oder 80 Bit IN st(0) laden
FILD int-Variable	Ganzzahl (Integer) mit 16, 32 oder 64 Bit laden
FBLD BCD-Variable	gepackte BCD-Zahl mit 80 Bit laden

Ebenfalls nützlich ist es, dass FLD auch auf Stackregister angewendet werden kann, z.B.:

FLD ST(2)

Dieser Befehl legt eine Kopie von ST(2) an die Spitze des Stack. Dabei wird der Wert in ST(0) zu ST(1) usw. ST(0) und ST(3) enthalten also nun den gleichen Wert. Den Datentransport von

den FPU-Registern in den Speicher nennt man *Speichern, Store* Die entsprechenden Befehle zum Abspeichern von Daten sind (ST=Store)

FST fp-Variable/Stackregister	ST(0) als Gleitkommazahl mit 32, 64 oder 80 Bit speichern
FIST int-Variable	ST(0) als Ganzzahl (Integer) mit 16, 32 oder 64 Bit speichern
FBST BCD-Variable	ST(0) als gepackte BCD-Zahl mit 80 Bit speichern

Bei diesen Speicher-Befehlen wird aber *keine* Zahl vom Stack entfernt, also kein implizites POP durchgeführt. Dies erreicht man erst durch die Befehlsvarianten mit dem angehängten P:

FSTP fp-Variable/Stackregister	Gleitkommazahlen mit 32, 64 oder 80 Bit speichern und POP
FISTP int-Variable	Ganzzahlen (Integer) mit 16, 32 oder 64 Bit speichern und POP
FBSTP BCD-Variable	gepackte BCD-Zahlen mit 80 Bit speichern und POP

FST und FSTP können auch auf Stackregister angewendet werden, womit der Wert des betreffenden Registers einfach überschrieben wird. So legt beispielsweise FSTP ST(3) ST(0) in ST(3) ab und führt anschließend ein pop durch. Danach ist also das bisherige ST(3) aus dem Stapel entfernt.

Ein nützlicher Befehl ist FXCH ST(i), der die Spitze des Stack ST(0) mit einem beliebigen Stackregister ST(i) austauscht.

Um vordefinierte Konstanten zu laden gibt es spezielle Ladebefehle, Beispiele sind:

FLDZ	Läd eine Null in ST(0)
FLD1	Läd eine eins in ST(0)
FLDPI	Läd π in ST(0)

11.3.2 Kontrollbefehle

Es gibt Kontrollbefehle um die FPU-Exceptions zu steuern, Steuer- und Statuswort im Speicher abzulegen, die Umgebung der FPU abzuspeichern oder zu laden, den Stackpointer zu manipulieren, Register explizit freizugeben, u.a.m. Beispiele:

FINIT	Software-Reset der FPU
FDECSTP	Dekrement des Stackregister Pointer (Top of Stack)
FINCSTP	Inkrement des Stackregister Pointer (Top of Stack)
FFREE	Register für frei d.h. ungültig erklären, verändert nicht den Stackpointer

11.3.3 Arithmetische Befehle

Gleitkommarechnungen benutzen die Register des Registerstacks und z.T. auch Speicheroperanden. Das wichtigste Register ist ST(0), es ist an allen Gleitkommabefehlen beteiligt. Es gibt auch die Möglichkeit, Speicheroperanden direkt in den Arithmetikbefehl einzubeziehen. Nehmen wir als Beispiel die Addition, es stehen folgende Varianten zur Verfügung:

FADD ST(0),ST(i)	ST(0) = ST(0)+ST(i)
FADD ST(i),ST(0)	ST(i) = ST(0)+ST(i)
FADDP ST(i),ST(0)	ST(i) = ST(0)+ST(i), pop stack
FADD fp-Variable	ST(0) = ST(0)+fp-Variable
FIADD int-Variable	ST(0) = ST(0)+int-Variable

Hierbei darf fp-Variable eine Gleitkomma-Speichervariable mit 32 oder 64 Bit sein und int-Variable eine Integer-Speichervariable mit 16 oder 32 Bit.

Einen Überblick über die arithmetischen Befehle und ihre Wirkung gibt die folgende Tabelle:

FADD	Ziel ← Ziel+Quelle
FSUB	Ziel ← Ziel-Quelle
FMUL	Ziel ← Ziel*Quelle
FDIV	Ziel ← Ziel/Quelle
FSUBR	Ziel ← Quelle-Ziel
FDIVR	Ziel ← Quelle/Ziel

Bei den beiden letzten Befehlen steht R für Reverse, weil die Operanden getauscht sind. Für alle Befehle in dieser Liste gibt es die oben gezeigten Varianten, also z.B. FMUL, FMULP, FIMUL usw. Dazu kommen weitere Befehle, wie z.B.

FABS	Bildet den Absolutwert von ST(0)
FCHS	Ändert das Vorzeichen von ST(0)
FSQRT	Zieht die Quadratwurzel aus ST(0)

11.3.4 Trigonometrische Befehle

Diese Befehle führen mächtige mathematische Berechnungen durch um trigonometrische, logarithmische oder exponentielle Funktionen zu berechnen. Vor Aufruf der trigonometrischen Funktionen muss das Argument im Bogenmass in ST(0) hinterlegt werden. Die Funktionen sind:

FSIN	ST(0)=Sinus (Argument),
FCOS	ST(0)=Cosinus (Argument)
FSINCOS	ST(0)=Sinus(Argument), ST(1)=Cosinus(Argument)
FPTAN	partieller Tangens: ST(0)=X, ST(1)=Y, Y/X=tan(Argument)
FPATAN	partieller Arcustangens, ST(0)=arctan(ST(1)/ST(0))

Für die Berechnung von Potenzen gibt es die folgenden Exponentialfunktionen

FYL2X	ST(0)= $Y * log_2 X$, Y=ST(1), X=ST(0)
FYL2XP1	ST(0)= $Y * log_2(X+1)$, Y=ST(1), X=ST(0)
F2XM1	ST(0)= $2^X - 1$, X=ST(0)
FSCALE	ST(0) = ST(0)*2^Y, Y=ST(1)

11.3.5 Vergleichsbefehle

COM=Compare.

FCOM ST(i)	Vergleiche ST(0) und ST(i)
FCOM fp-Variable	Vergleiche ST(0) und fp-Variable
FICOM int-Variable	Vergleiche ST(0) und int-Variable
FCOMI fp-variable	vergleiche ST(0) und fp-Variable und setze EFlags für Jcc-Befehle

Zusätzlich gibt es die Varianten FCOMP und FICOMP die nach dem Vergleich ST(0) vom Stack entfernen, FCOMPP und FICOMPP entfernen ST(0) und ST(1) vom Stack.

In einem abschließenden Beispiel soll mit einer einfachen Berechnung die Benutzung der Gleitkommaeinheit gezeigt werden. Dabei wird die Höhe eines senkrecht geworfenen Körpers ohne Luftreibung berechnet. Diese ist bekanntlich durch

$$h = v_0 t - \frac{1}{2} g t^2$$

gegeben, wenn v_0 die Anfangsgeschwindigkeit, t die Zeit und g die Erdbeschleunigung ist. Das Programm benutzt nur die beiden ersten Register des Gleitkommastacks st(0) und st(1). Für die Berechnung wird die obige Formel geschrieben als $h = (v_0 - \frac{1}{2} g t)t$

```
;   Berechnung der Hoehe eines Körpers beim senkrechten
;   Wurf nach oben mit der Gleitkommaeinheit
   .MODEL SMALL     ; Speichermodell "SMALL",
   .STACK 100h      ; 256 Byte Stack reservieren
   .DATA
; Fliesskommavariablen mit 64 Bit
v0      dq  13.5
g       dq  9.81
einhalb dq  0.5
t       dq  1.3
Hoehe    dq  ?
    .CODE
    .386
    .387
Programmstart:
    mov ax,@data
    mov ds,ax
;
; senkrechter Wurf nach oben:
; h = (v0 - 0.5*g*t) *t
;
    finit
    fld t                   ; t in st(0)
    fst st(1)               ; Kopie nach st(1)
    fmul g                  ; st(0)=g*t
    fmul einhalb            ; st(0)=0.5*g*t
    fsubr v0                ; st(0)=v0 - 0.5*g*t
    fmulp st(1),st(0)       ; st(1)=st(1)*st(0), pop
    fstp Hoehe              ; zurueckschreiben auf Variable Hoehe
EXIT:
   mov  ah,04Ch              ; ah=04C : DOS-Funktion "terminate the program"
   mov  al,0                 ; DOS-Return-Code
   int  21h                  ; Interrupt 21h : Aufruf von DOS
END Programmstart        ; Ende Uebersetzung
```

Kapitel 12

Die MMX-Einheit

12.1 SIMD, Sättigungsarithmetik und MAC-Befehle

In den 90er Jahren wurde klar, dass PC's in immer neue Anwendungsbereiche vordringen, und dass dazu auch Bearbeitung und Wiedergabe von Audio- und Videodaten gehört. Dazu müssen meist sehr viele kleine Dateneinheiten verarbeitet werden, die in Feldern liegen oder kontinuierlich angeliefert werden. So wird ein Lautstärkewert (Abtastwert) in CD-Qualität durch 2 Byte (16 Bit) dargestellt, ein Grauwert oft durch 1 Byte, ein Farbwert durch 3 Byte. Die Prozessor-Register mit 32 oder mehr Bit sind also schlecht ausgenutzt. Die Registerausnutzung wäre mit *gepackten Dateneinheiten* besser, z.B. ein 32 Bit-Register als zwei unabhängige Worte oder vier unabhängige Byte zu behandeln. Wenn man dann diese unabhängigen Teildaten gleichzeitig bearbeiten könnte, hätte man natürlich viel Zeit gewonnen. Diese Idee ist schon alt und wird als *Single Instruction - Multiple Data* (*SIMD*) bezeichnet.

Eine andere Problematik ist, dass Standardarithmetik der ALU oft für die Bearbeitung von Audio- und Videodaten unpraktisch ist. Die Erhöhung des Helligkeitswertes 255 um 1 führt in 8-Bit-Darstellung durch den Register-Überlauf zu einem Helligkeitswert 0, der sicher nicht erwünscht ist. Ebenso würde eine Verringerung eines Helligkeitswertes 0 um 1 zu einem Helligkeitswert von 255 führen, was ebenfalls keinen Sinn macht. Besser wären Sättigungsendwerte von 255 und 0, also

Sättigung am oberen Endwert: 255+1=255, 255+2=255 usw.

Sättigung am unteren Endwert: 0-1=0, 0-2=0 usw.

Feste Endwerte statt Überlauf und Unterlauf nennt man auch *Sättigungsarithmetik* .

Eine weitere für Multimedia wünschenwerte Eigenschaft ist ein Multiply and Accumulate-Befehl (MAC-Befehl), der in der digitalen Signalverarbeitung ständig gebraucht wird. MAC-Befehle multiplizieren zwei Zahlen und addieren das Ergebnis zu einer mitgeführten Summe, mehr dazu weiter unten.

12.2 Register, Datenformate und Befehle

Die 1997 eingeführte MMX-Einheit der Pentium-Prozessoren (**M**ulti**m**edia E**x**tension bietet nun genau diese Spezialfeatures: SIMD, Sättigungsarithmetik und auch einen MAC-Befehl. Es stehen acht MMX-Register mit je 64 Bit Breite zur Verfügung. Diese Register können wie folgt Daten austauschen:

- In 32 Bit Breite mit den Universalregistern und dem Speicher, Befehl `movd` (move doubleword)
- In 64 Bit Breite mit dem Speicher, Befehl `movq` (move quadword)

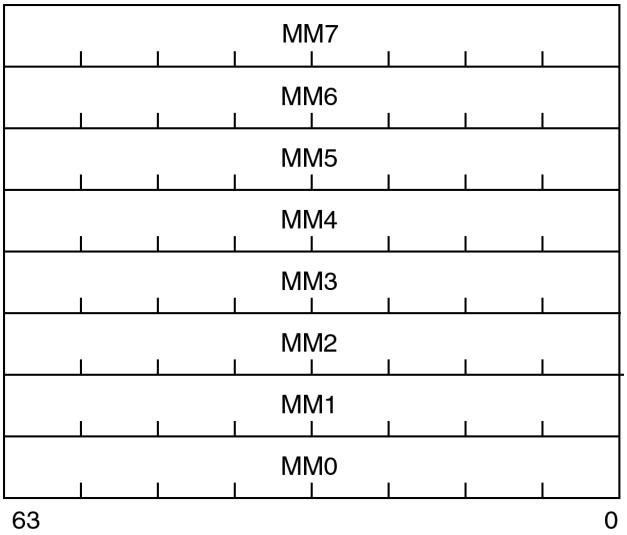


Abbildung 12.1: Die MMX-Einheit verfügt über acht Register, alle mit einer Breite von 64 Bit.

Man muss erwähnen, dass die MMX-Einheit nicht wirklich acht neue Register hat, sondern dass diese Register die Mantissenanteile der Gleitkommaregister sind. Dies ist wichtig für die Programmierung: Ein Schreibvorgang auf ein MMX-Register zerstört FPU-daten und umgekehrt. Man sollte daher MMX- und FPU-Befehle nicht mischen und muss nach Abschluss der MMX-Operationen mit dem Befehl `emms` (empty multimedia state) die Register leer hinterlassen (FPU-Tag-Worte alle gleich 11b).

Ausserdem werden drei neue Formate definiert, die gepackte Ganzzahlen enthalten und mit 64 Bit Breite natürlich genau in die MMX-Register passen. Zusätzlich kann die MMX-Einheit auch eine ungepackte 64-Bit-Ganzzahl in einem MMX-Register ablegen.

An einem Beispiel soll nun eine typische SIMD-Operation auf einer MMX-Einheit gezeigt werden. Der Befehl `paddb` *Operand1, Operand2* behandelt die beiden Operanden als Einheit von acht gepackten Bytes, führt acht paarweise Additionen aus und speichert die acht Ergebnisse im ersten Operanden ab. Das Schema ist in Abb.12.2 gezeigt.

In dem folgenden Programmierbeispiel wird nur die untere Hälfte der MMX-Register ausgenutzt:

```
mov eax, 010203F0h
```

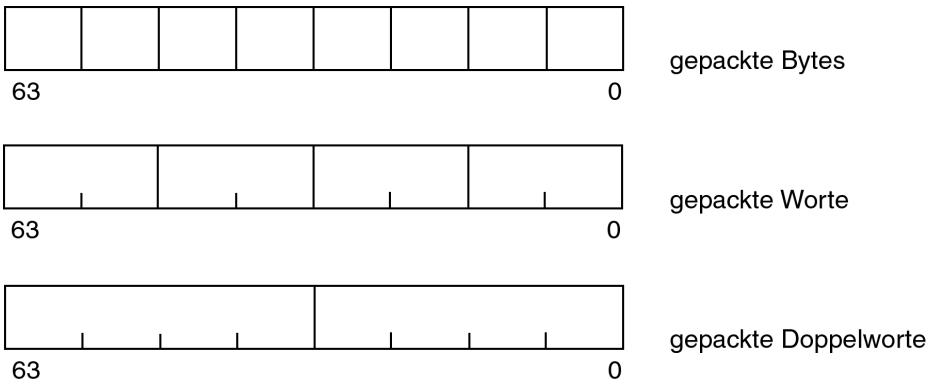


Abbildung 12.2: Die gepackten MMX-Formate.

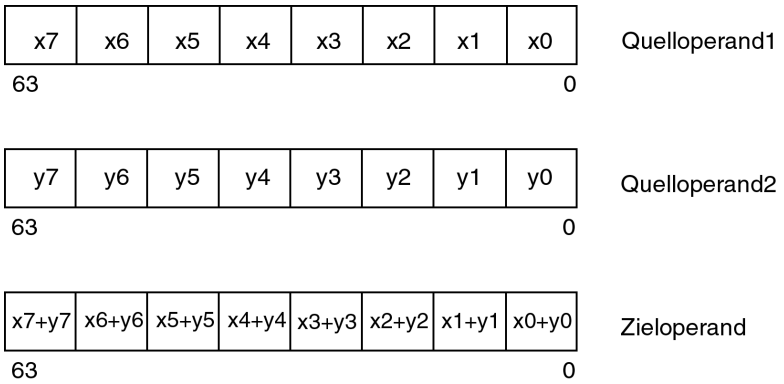


Abbildung 12.3: Die Addition von acht gepackten Bytes mit dem PADDB-Befehl.

```
mov ebx, 02030410h  ; Summe bei 32-Bit-Ganzzahladdition ist 03050800h.
movd mm0, eax
movd mm1, ebx
paddb mm0,mm1       ; gepackte Addition von Bytes
movd eax,mm0        ; Ergebnis: EAX=03050700h
```

Der Unterschied zur 32-Bit-Ganzzahlarithmetik liegt in der Stelle mit der Ziffer "7", der Übertrag aus der Addition der unteren Bytes wurde eben nicht auf das nächste Byte übertragen, da die acht Bytes als unabhängige Einheiten behandelt werden! Der Befehl `paddw` addiert vier gepackte Worte, `paddd` addiert zwei gepackte Doppelworte. Betrachten wir noch einmal das obige Beispiel: Auf dem unteren Byte findet genau der im Multimediabereich unsinnige Überlauf statt, den man mit Sättigungsarithmetik vermeiden kann.

```
mov eax, 010203F0h
mov ebx, 02030410h  ; Summe bei 32-Bit-Ganzzahladdition ist 03050800h.
movd mm0, eax
movd mm1, ebx
paddusb mm0,mm1     ; gepackte Addition von Bytes, vorzeichenlose Sättigung
movd eax,mm0        ; Ergebnis: EAX=030507FFh
```

`paddusb` steht für "packed add with unsigned saturation bytes". Das Ergebnis FFh im letzten Byte ist der Sättigungswert vorzeichenloser 8-Bit-Zahlen. In Tabelle 12.1 sind alle Sättigungswerte

		unterer Sättigungswert	oberer Sättigungswert
Vorzeichenlose Zahl	8 Bit	0	255
	16 Bit	0	65535
Vorzeichenbehaftete Zahl	8 Bit	-128	127
	16 Bit	-32768	32767

Tabelle 12.1: Die Endwerte der Sättigungsarithmetik

zusammengestellt.

12.3 Der PMADDWD-Befehl: Unterstützung der digitalen Signalverarbeitung

Um den Sinn des Befehls `PMADDWD` zu verstehen machen wir einen kleinen Abstecher in die digitale Signalverarbeitung (DSV). Ein digitales System verarbeitet eine Eingangszahlenfolge

$$x(n), x(n-1), x(n-2) \ldots$$

wobei $x(n)$ der letzte Eingangswert ist, $x(n-1)$ der vorletzte Eingangswert usw. Diese Eingangswerte könnten z.B. digitalisierte Signalamplituden von einem Mikrofon, einem Modem oder einem Tonträger sein. Sie könnten aber auch die Helligkeitswerte einer Bitmap sein. Das digitale System ermittelt daraus eine Ausgangszahlenfolge

$$y(n), y(n-1), y(n-2) \ldots$$

Dieses Signal kann digital weiterverarbeitet werden oder aber in ein analoges Signal umgesetzt werden. Es wird also aus dem Eingangssignal ein verändertes Ausgangssignal gemacht. Die Rechenvorschrift, nach der die digitale Verarbeitung des Signals erfolgt, lautet:

$$\begin{aligned} y(n) &= A_0 x(n) + A_1 x(n-1) + \ldots + A_m x(n-m) \\ &\quad -B_1 y(n-1) - B_2 y(n-2) - \ldots - B_k y(n-k) \end{aligned} \tag{12.1}$$

Ein digitales signalverarbeitendes System wird also beschrieben durch die Koeffizienten

$$A_0 \ldots A_m, B_1 \ldots B_k$$

Sind alle $B_i = 0$ so handelt es sich um ein System mit endlicher Impulsantwort, einen FIR-Filter (Finite Impulse Response). Ist dagegen mindestens ein $B_i \neq 0$, so handelt es sich um ein System mit unendlicher Impulsantwort. Ein Beispiel: Durch $A_1 = 1, A_2 = 2, A_3 = 1, B_1 = 1.937, B_2 = -0.9400$ wird ein IIR-Filter (hier ein Tschebischeff-Tiefpass-Filter zweiter Ordnung) dargestellt.

Diese Koeffizientensätze können recht lang sein, besonders bei FIR-Filtern; 50 Koeffizienten sind nicht ungewöhnlich. Wird das signalverarbeitende System mit einem Mikroprozessor realisiert,

so muss obige Rechenvorschrift in ein Programm umgesetzt werden. Bei einem digitalen System mit 50 Koeffizienten muss dazu 50 mal das Produkt $A_m x(n-m)$ gebildet und zu der Zwischensumme addiert werden, um die Summe und damit einen einzigen Ausgangswert zu errechnen! Der Rechenaufwand ist also hoch und muss noch dazu in Echtzeit bewältigt werden, um nicht den Anschluss an den Datenstrom zu verlieren. Hier hilft ein Befehl, der eine Multiplikation und Addition des Ergebnisses in einem Schritt ausführt, ein sog. *MAC-Befehl* (Multiply and accumulate).

Der Befehl `PMADDWD` der MMX-Einheit führt sogar zwei zweifache MAC-Operationen in einem Schritt parallel aus. Er arbeitet nach folgendem Schema: Auch hierzu ein Programmbeispiel,

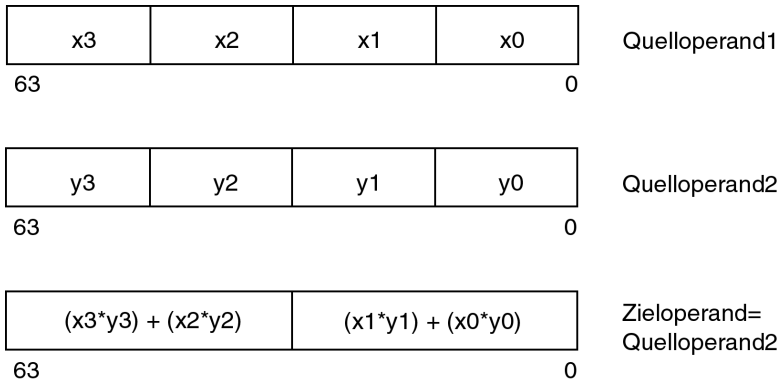


Abbildung 12.4: Der PMADDWD-Befehl multipliziert vier Paare von Worten und addiert paarweise die Produkte auf, er ist ein zweifach paralleler MAC-Befehl

das allerdings der Einfachheit halber nur die untere Hälfte der Register ausnutzt, also nur einen MAC-Befehl durchführt:

```
mov eax, 00010004h
mov ebx, 00030002h  ;
movd mm0, eax
movd mm1, ebx
pmaddwd mm0,mm1     ; Mutliplikationen des unteren Wortpaares (2*4),
                    ; des oberen Wortpaares (1*3) und Addition (3+8)
movd eax,mm0        ; Ergebnis: EAX=0000000Bh=11d
```

Um nun wirklich ein digitales signalverarbeitendes System zu realisieren, müßte man zunächst die Eingangswerte und Koeffizienten so skalieren, dass man mit 16-Bit-Worten ausreichende Genauigkeit erreicht. Dann könnte man z.B. in mm1 den Eingangswert $x(n-m)$ und die Zwischensumme ablegen, in mm0 dagegen a_m und eine 1. Der Befehl `pmaddwd` würde dann in einem Schritt $Zwischensumme + x(n-m)a_m$ berechnen und in der unteren Hälfte von mm0 ablegen. In der oberen Hälfte der Einheit könnte gleichzeitig etwas anderes berechnet werden, z.B. eine neue Zwischensumme aus dem $B_k y(n-k)$-Zweig.

12.4 Befehlsübersicht

Zum Schluss des Kapitels gebe ich noch eine knappe Übersicht über den Befehlssatz der MMX-Einheit:

Arithmetische Befehle Addition, Subtraktion, und Multiplikation gepackter Daten mit und ohne Sättigung, Multiplikation und Addition in einem Schritt

Vergleichsbefehle Vergleich gepackter Daten auf Gleichheit oder größer, Ablage der Ergebnisflags im Zielregister

Umwandlungsbefehle Packen von Worten in Bytes oder Doppelworten in Worte, dabei Befolgung der Sättigungsarithmetik

Entpackbefehle Erweiterung von Bytes zu Worten, Worten zu Doppelworten und Doppelworten zu Quadworten (64-Bit).

Bitweise logische Befehle Logisches UND, UND NICHT, ODER und exklusives ODER auf 64-Bit-Operanden

Schiebebefehle Schieben der gepackten Worte, Doppelworte oder Quadwortes (ganzes 64-Bit-Register) nach links oder rechts.

Datentransport Bewegen eines Doppelwortes zwischen MMX-Einheit und Speicher/Allzweckregister und Bewegen eines Quadwortes zwischen MMX-Einheit und Speicher

EMMS Beenden der Multimediaoperationen und Leeren der MMX-Register

Kapitel 13

Die Schnittstelle zwischen Assembler und C/C++

13.1 Übersicht

Man verwendet Assembler in C/C++ (oder anderen Hochsprachen) aus zwei Gründen:

Geschwindigkeitsgewinn Häufig durchlaufene Codeabschnitte können zeitkritischen sein, z.B. der Kern eines MP3-Dekoders oder eines Spieles. Es gibt Hilfsmittel um zeitkritische Abschnitte zu identifizieren. Dort kann es sich lohnen, Assembler einzusetzen. Man würde das Programm zunächst in C/C++ schreiben und dann die wichtigsten Stellen, so sparsam wie möglich (z.B. 2%) durch Assemblercode ersetzen. Die ursprünglichen C/C++ Befehle sollten als Kommentar stehen bleiben. Nachteile: Portabilität geht verloren, Wartung wird schwierger, Lesbarkeit schlechter.

Vollständiger Zugriff auf Prozessor und Hardware Mit Assembler können z.B. Flags manipuliert oder Stringbefehle erzwungen werden. Auch die Prozessorkontrollbefehle für den Protected Mode können nur in Assembler eingefügt werden. Praktisch ist Assembler auch beim Zugriff auf IO-Ports. Alle direkten Hardwarezugriff sind unter Multiuser-Betriebssystemen allerdings dem Betriebssystem und den Geräte-Treibern vorbehalten.

Für die Anwendung von Assembler in C/C++ gibt es zwei Möglichkeiten:

Inline-Assembler Eine sehr einfache Möglichkeit: Statt eines C/C++ Befehles kann jederzeit ein Assemblerbefehl oder ein Block aus Assemblerbefehlen stehen; es ergibt sich also eine Mischung aus C/C++ und eingestreuten Assemblerbefehlen.

Externe Assemblerprogramme Unterprogramme werden separat assembliert und dann zu dem kompilierten C/C++-Programm gebunden. Externe Assemblerprogramme müssen sich streng an die Konventionen der Hochsprache halten, z.B. Stack-Übergabe von Parametern und Rückgabe von Ergebnissen in vereinbarten Registern. (gelinkt).

13.2 16-/32-Bit-Umgebungen

	16-Bit-Umgebung	**32-Bit-Umgebung**
Bsp. Betriebssystem Prozessor	DOS (Real Mode) 8086 - Pentium	Windows 98/NT, Linux 80386 - Pentium
typ. Registerbenutzung typ. Assemblerbefehle	AX,BX,CX, ... 8086-Befehlssatz	EAX, EBX, ECX, ... 80386-Befehlssatz
Zeiger	NEAR PTR: Offset zu 16 Bit FAR PTR: Segment, Offset je 16 Bit (zwei Register)	32-Bit-Offset (ein Register)
Speicheraufbau	Segmente zu 64 kB	flaches Speichermodell alles liegt in einem Segment von max. 4GB
Physikalische Adressen	16*Seg + Offs für Programmierer ausrechenbar	vom Betriebssystem verwaltet für Programmierer unbekannt Paging
C-Datentyp char, unsigned char C-Datentyp short, unsigned short C-Datentyp int, unsigned int C-Datentyp long, unsigned long C-Datentyp float C-Datentyp double	8 Bit 16 Bit 16 Bit 32 Bit 32 Bit 64 Bit	8 Bit 16 Bit 32 Bit 32 Bit 32 Bit 64 Bit
Push, Pop	16 Bit	32 Bit
Betriebssystemanbindung	Int 21h, Intxxh	Call Bibliotheksfunktion

13.3 Aufbau und Funktion des Stack

Der Stack dient in C-Programmen drei Zwecken:

- Übergabe von Parametern
- Speicherung der Rücksprungadresse bei Funktionsaufrufen
- Speicherung von lokalen Variablen

Beim Aufruf einer Funktion kommen diese Daten auch gerade in dieser Reihenfolge auf den Stack. Die einzelnen Schritte sind können in dem Beispielprogramms auf im folgenden Abschnitt sehr schön im Code verfolgt werden und sind dort auch kommentiert. Die nachfolgend dargestellten Punkte sind im Code und den Kommentaren zu den Zeilen 3, 7, 22, 23 und 28 im Detail zu sehen.

Parameterablage Das rufende Programm legt zunächst die zu übergebenden Parameter – standardmäßig von rechts nach links – auf dem Stack ab (Code und Kommentar zu Zeilen 22 und 23).

Funktionsaufruf Der Aufruf der Funktion wird durch CALL durchgeführt, dabei legt der Prozessor die Rücksprungadresse auf den Stack (Code und Kommentar zu Zeilen 22 und 23).

EBP sichern Die gerufene Funktion selbst sichert nun zunächst den aktuellen Inhalt von EBP auf dem Stack (PUSH EBP)

ESP auf EBP kopieren Um den Stack über EBP bequem adressieren zu können und ESP frei zu haben für weitere Stackreservierungen (z.B. durch PUSH) wird ESP auf EBP kopiert.

Platz für lokale Variable reservieren Durch Verkleinerung von ESP oder durch PUSH kann für benötigte lokale Variable Platz auf dem Stack reserviert werden (Code und Kommentar zu Zeile 3).

Freigabe der lokalen Variablen Durch Zurücksetzen des ESP auf den Wert vor der Reservierung wird vor Beendigung der Funktion der Speicherplatz der lokalen Variablen wieder freigegeben (Zeile 7). Der gerettete EBP-Inhalt muss jetzt Top of Stack (TOS) sein.

EBP wieder herstellen Durch POP EBP erhält EBP wieder den Wert, den er vor dem Funktionsaufruf hatte. Erst dadurch sind geschachtelte Funktionsaufrufe möglich!

Rücksprung TOS ist jetzt die Rücksprungadresse, die durch RET (Return) vom Stack genommen wird.

Parameter vom Stack entfernen Nun müssen nur noch die übergebenen Parameter vom Stack entfernt werden. Dies wird bei standardmäßiger C-Kompilierung durch das rufende Programm ausgeführt.

Die Art der Parameterübergabe und -entfernung lässt sich durch Compileroptionen steuern (s.Abschn.13.5.1) Wenn man beispielhaft annimmt, dass die Parameter und die lokalen Variablen 4 Byte groß sind, hat durch dieses Vorgehen der Stack während der Ausführung einer Funktion folgenden Aufbau, der auch als sog. *Stackframe* bekannt ist. (Ein Stackframe in einem 16-Bit-Programm ist in Abschnitt 13.6.3 gezeigt.)

Adresse	Inhalt
`EBP + 20`	.
`EBP + 16`	.
`EBP + 12`	zweites Parameterwort
`EBP + 8`	erstes Parameterwort
`EBP + 4`	Rücksprungadresse
`EBP`	geretteter `EBP`
`EBP - 4`	erstes Wort der lokalen Variablen
`EBP - 8`	zweites Wort der lokalen Variablen
`EBP - 12`	drittes Wort der lokalen Variablen
`EBP - 16`	.
`EBP - 20`	.

13.4 Erzeugung von Assemblercode durch Compiler

Das folgende Beispiel zeigt ein C++-Programm, das mit einem 32-Bit-Compiler übersetzt wurde. Zunächst der C++-Quellcode:

```
#include <stdio.h>

int Produkt(int Faktor1, int Faktor2) {
int Ergebnis;
Ergebnis = Faktor1 * Faktor2;
return Ergebnis;
}

main() {
char ch;
int a,b,c;
int *p1, *p2;
float f=15.0;

// Benutzung von Zeigern
p1 = &a;        // p1 erhält die Adresse von a
*p1 = 100;      // a=100
p2 = p1;        // p2 enthält die gleiche Adresse wie p1
printf("p2 zeigt auf die Variable a=%i, *p2=%i \n",a,*p2);

b=25;
c=Produkt(a,b);     // Funktionsaufruf
printf("%6i\n",c);

f=f/2;  // Fliesskomma-Operationen

return 0;
}
```

Der Compiler erzeugt aus diesem Quellcode folgenden Assemblercode:

```
--- C:\ASM\inlasm\codebsp1.cpp  ----------------------------------------------------------------------
1:    #include <stdio.h>
2:
3:    int Produkt(int Faktor1, int Faktor2) {
00401020 55                   push        ebp
00401021 8B EC                mov         ebp,esp
00401023 51                   push        ecx
4:    int Ergebnis;
5:    Ergebnis = Faktor1 * Faktor2;
00401024 8B 45 08             mov         eax,dword ptr [Faktor1]
00401027 0F AF 45 0C          imul        eax,dword ptr [Faktor2]
0040102B 89 45 FC             mov         dword ptr [Ergebnis],eax
6:    return Ergebnis;
0040102E 8B 45 FC             mov         eax,dword ptr [Ergebnis]
7:    }
00401031 8B E5                mov         esp,ebp
00401033 5D                   pop         ebp
00401034 C3                   ret
8:
9:
10:   main() {
00401035 55                   push        ebp
```

```
00401036 8B EC               mov         ebp,esp
00401038 83 EC 18            sub         esp,18h
11:   int a,b,c;
12:   int *p1, *p2;
13:   float f=15.0;
0040103B C7 45 F0 00 00 70 41 mov         dword ptr [f],41700000h
14:
15:   // Benutzung von Zeigern
16:   p1 = &a;        // p1 erhält die Adresse von a
00401042 8D 45 FC            lea         eax,dword ptr [a]
00401045 89 45 EC            mov         dword ptr [p1],eax
17:   *p1 = 100;      // a=100
00401048 8B 4D EC            mov         ecx,dword ptr [p1]
0040104B C7 01 64 00 00 00    mov         dword ptr [ecx],64h
18:   p2 = p1;        // p2 enthält die gleiche Adresse wie p1
00401051 8B 55 EC            mov         edx,dword ptr [p1]
00401054 89 55 E8            mov         dword ptr [p2],edx
19:   printf("p2 zeigt auf die Variable a=%i, *p2=%i \n",a,*p2);
00401057 8B 45 E8            mov         eax,dword ptr [p2]
0040105A 8B 08               mov         ecx,dword ptr [eax]
0040105C 51                  push        ecx
0040105D 8B 55 FC            mov         edx,dword ptr [a]
00401060 52                  push        edx
00401061 68 30 5A 41 00       push        offset ___xt_z(0x00415a30)+10Ch
00401066 E8 65 00 00 00       call        printf(0x004010d0)
0040106B 83 C4 0C            add         esp,0Ch
20:
21:   b=25;
0040106E C7 45 F8 19 00 00 00 mov         dword ptr [b],19h
22:   c=Produkt(a,b);     // Funktionsaufruf
00401075 8B 45 F8            mov         eax,dword ptr [b]
00401078 50                  push        eax
00401079 8B 4D FC            mov         ecx,dword ptr [a]
0040107C 51                  push        ecx
0040107D E8 7E FF FF FF       call        @ILT+0(?Produkt@@YAHHH@Z)(0x00401000)
00401082 83 C4 08            add         esp,8
00401085 89 45 F4            mov         dword ptr [c],eax
23:   printf("%6i\n",c);
00401088 8B 55 F4            mov         edx,dword ptr [c]
0040108B 52                  push        edx
0040108C 68 5C 5A 41 00       push        offset ___xt_z(0x00415a5c)+138h
00401091 E8 3A 00 00 00       call        printf(0x004010d0)
00401096 83 C4 08            add         esp,8
24:
25:   f=f/2;  // Fliesskomma-Operationen
00401099 D9 45 F0            fld         dword ptr [f]
0040109C D8 35 54 30 41 00    fdiv        dword ptr [??_C@_08GNFC@printf?4c?$AA@(0x00413054)-4]
004010A2 D9 5D F0            fstp        dword ptr [f]
26:
27:   return 0;
004010A5 33 C0               xor         eax,eax
28:   }
004010A7 8B E5               mov         esp,ebp
004010A9 5D                  pop         ebp
```

```
004010AA C3                   ret
--- Ende Quellcodedatei  ------------------------------------------------------------------------
```

Zum Verständnis dieses Assemblercodes die folgenden Kommentare. Sie beziehen sich auf die Assemblerbefehle, durch die der C++-Befehl in der genannten Zeile realisiert wird. Man sieht sehr deutlich, dass es sich um 32-Bit-Code handelt: Es werden 32-Bit-Register benutzt (EAX,EBX usw.), Zeiger sind einfache 32-Bit-Zahlen, Integer sind mit 32-Bit codiert, die Register ecx und eax werden für die Adressierung benutzt (nicht möglich beim i8086).

Zeile 3 Aufbau des Stackframes mit ebp und esp als 32-Bit-Zeigerregister Reservierung von 32 Bit für die lokale Variable Ergebnis durch push ecx

Zeile 4,5 Multiplikation von Faktor1 und Faktor2, Resultat in Ergebnis abspeichern. Faktor1, Faktor2 und Ergebnis sind vom Typ Integer und werden als 32-Bit-Variablen auf dem Stack angelegt.

Zeile 6 Rückgabe des Ergebnisses in EAX

Zeile 7 Durch "}"wird die Funktion beendet: Abbau Stackframe und RET-Befehl

Zeile 10 – 13 „main“ wird wie jede andere Funktion übersetzt, Stackframe und Platz für lokale Variable auf dem Stack: drei Integer zu je 4 Byte, zwei Zeiger zu je 4 Byte, eine float-Var. zu 4 Byte ergeben 24 Byte (18h), daher also: sub esp,18h; Initialisierung von f mit 32-Bit

Zeile 16 Adresse (Offset) von a via EAX nach p1 kopieren.

Zeile 17 Der Wert 100 (64h) wird auf den Speicherplatz geschrieben, dessen Adresse in p1 steht.

Zeile 18 Kopieren der Adresse in p1 via EDX nach p2

Zeile 19 Die drei Argumente des printf-Aufrufs werden – beginnend mit dem letzten – nacheinander auf den Stack gebracht; Aufruf der Bibliotheks-Prozedur printf, anschliessende Stackbereinigung (12 Byte)

Zeile 21 Wert 25 (19h) in Variable b

Zeile 22 a und b auf den Stack bringen, Aufruf der selbstgeschriebenen Prozedur Produkt, acht Byte wieder vom Stack entfernen, Funktionsergebnis aus EAX entnehmen und in c kopieren

Zeile 23 zwei Parameter auf Stack bringen, Aufruf von printf, Rückgabewert wird nicht verwertet, acht Byte vom Stack entfernen

Zeile 25 Division einer Fließkommazahl durch drei Koprozessorbefehle: fld (Laden der Variablen), fdiv (Division), fstp (Speichern)

Zeile 27 Rückgabewert von “main“ ist Null und kommt nach EAX.

Zeile 28 Abbau des Stackframes, RET-Befehl

13.5 Steuerung der Kompilierung

13.5.1 Aufrufkonventionen

Aufrufkonventionen bestimmen, wie die Parameterübergabe an Funktionen gestaltet wird. In dem folgenden Beispiel wird eine Funktion mit drei verschiedenen Aufrufkonventionen übersetzt:

```
; Beispiel für die Wirkung von Aufrufkonventionen
; (Segment- und andere Direktiven weggelassen)


; default-Aufrufkonvention (_cdecl)
; - Stack-Parameterübergabe,
; - Reihenfolge von rechts nach links,
; - aufrufende Funktion räumt Stack auf

_a$ = 8
_b$ = 12
_aminusb$ = -4
?idifferenz1@@YAHHH@Z PROC NEAR             ; idifferenz1

; 3    : int idifferenz1(int a,int b) {
    push    ebp
    mov ebp, esp
    push    ecx
; 4    : int aminusb;
; 5    : aminusb = a - b;
    mov eax, DWORD PTR _a$[ebp]
    sub eax, DWORD PTR _b$[ebp]
    mov DWORD PTR _aminusb$[ebp], eax
; 6    : return aminusb;
    mov eax, DWORD PTR _aminusb$[ebp]
; 7    : }
    mov esp, ebp
    pop ebp
    ret 0
?idifferenz1@@YAHHH@Z ENDP              ; idifferenz1


; Aufrufkonvention _stdcall
; - Stack-Parameterübergabe,
; - Reihenfolge von rechts nach links,
; - aufgerufene Funktion räumt Stack auf

_a$ = 8
_b$ = 12
_aminusb$ = -4
?idifferenz2@@YGHHH@Z PROC NEAR             ; idifferenz2

; 9    : int _stdcall idifferenz2(int a,int b) {
    push    ebp
    mov ebp, esp
```

```
    push    ecx
; 10   : int aminusb;
; 11   : aminusb = a - b;
    mov eax, DWORD PTR _a$[ebp]
    sub eax, DWORD PTR _b$[ebp]
    mov DWORD PTR _aminusb$[ebp], eax
; 12   : return aminusb;
    mov eax, DWORD PTR _aminusb$[ebp]
; 13   : }
    mov esp, ebp
    pop ebp
    ret 8
?idifferenz2@@YGHHH@Z ENDP              ; idifferenz2


; Aufrufkonvention _fastcall
; - Register-Parameterübergabe,
; - Reihenfolge von rechts nach links,
; - aufrufende Funktion räumt Stack auf

_a$ = -8
_b$ = -12
_aminusb$ = -4
?idifferenz3@@YIHHH@Z PROC NEAR             ; idifferenz3

; 15   : int _fastcall idifferenz3(int a,int b) {
    push    ebp
    mov ebp, esp
    sub esp, 12                 ; 0000000cH
    mov DWORD PTR _b$[ebp], edx
    mov DWORD PTR _a$[ebp], ecx
; 16   : int aminusb;
; 17   : aminusb = a - b;
    mov eax, DWORD PTR _a$[ebp]
    sub eax, DWORD PTR _b$[ebp]
    mov DWORD PTR _aminusb$[ebp], eax
; 18   : return aminusb;
    mov eax, DWORD PTR _aminusb$[ebp]
; 19   : }
    mov esp, ebp
    pop ebp
    ret 0
?idifferenz3@@YIHHH@Z ENDP              ; idifferenz3


_i$ = -4
_j$ = -8
_k$ = -12
_main   PROC NEAR

; 21   : main() {
    push    ebp
    mov ebp, esp
    sub esp, 12                 ; 0000000cH
```

```
; 22   : int i, j, k;
; 23   :
; 24   : i=100;
    mov DWORD PTR _i$[ebp], 100         ; 00000064H
; 25   : j=1;
    mov DWORD PTR _j$[ebp], 1
; 26   :
; 27   : k=idifferenz1(i,j);
    mov eax, DWORD PTR _j$[ebp]
    push    eax
    mov ecx, DWORD PTR _i$[ebp]
    push    ecx
    call    ?idifferenz1@@YAHHH@Z           ; idifferenz1
    add esp, 8
    mov DWORD PTR _k$[ebp], eax
; 28   :
; 29   : k=idifferenz2(i,j);
    mov edx, DWORD PTR _j$[ebp]
    push    edx
    mov eax, DWORD PTR _i$[ebp]
    push    eax
    call    ?idifferenz2@@YGHHH@Z           ; idifferenz2
    mov DWORD PTR _k$[ebp], eax
; 30   :
; 31   : k=idifferenz3(i,j);
    mov edx, DWORD PTR _j$[ebp]
    mov ecx, DWORD PTR _i$[ebp]
    call    ?idifferenz3@@YIHHH@Z           ; idifferenz3
    mov DWORD PTR _k$[ebp], eax
; 32   :
; 33   : return 0;
    xor eax, eax
; 34   :
; 35   : }
    mov esp, ebp
    pop ebp
    ret 0
_main   ENDP
END
```

13.5.2 Optimierungen

In dem folgenden Beispiel wird die Übersetzung des gleichen Programms optimiert auf hohe Ausführungsgeschwindigkeit. Der Compiler versucht dann überflüssige Befehle wegzulassen, die z.B. in obigem Beispiel bei der Registerübergabe offensichtlich sind.

```
; Der gleiche Code nach optimierter Ü\-ber\-set\-zung (Option /O2)
; (Segmentdirektiven weggelassen)

_a$ = 8
_b$ = 12
?idifferenz1@@YAHHH@Z PROC NEAR             ; idifferenz1, COMDAT
```

```
; 4    : int aminusb;
; 5    : aminusb = a - b;           ; lokale Variable aminusb nicht angelegt!
                                     ; dadurch bleibt esp unverändert
    mov eax, DWORD PTR _a$[esp-4]    ; Stackframe eingespart, Adressierung mit ESP
    mov ecx, DWORD PTR _b$[esp-4]    ; da dies hier möglich ist (ESP unverändert)
    sub eax, ecx
; 6    : return aminusb;
; 7    : }
    ret 0
?idifferenz1@@YAHHH@Z ENDP              ; idifferenz1


_a$ = 8
_b$ = 12
?idifferenz2@@YGHHH@Z PROC NEAR             ; idifferenz2, COMDAT
; 10   : int aminusb;
; 11   : aminusb = a - b;
    mov eax, DWORD PTR _a$[esp-4]       ; ähnlich idifferenz1
    mov ecx, DWORD PTR _b$[esp-4]
    sub eax, ecx
; 12   : return aminusb;
; 13   : }
    ret 8
?idifferenz2@@YGHHH@Z ENDP              ; idifferenz2


?idifferenz3@@YIHHH@Z PROC NEAR             ; idifferenz3, COMDAT

; 15   : int _fastcall idifferenz3(int a,int b) {

    mov eax, ecx                ; Registerübergabe,
                                ; kein Stackframe, keine lokale Variable
; 16   : int aminusb;
; 17   : aminusb = a - b;
    sub eax, edx
; 18   : return aminusb;
; 19   : }
    ret 0
?idifferenz3@@YIHHH@Z ENDP              ; idifferenz3



_main   PROC NEAR                   ; COMDAT
; 22   : int i, j, k;
; 23   :
; 24   : i=100;
; 25   : j=1;
; 26   :
; 27   : k=idifferenz1(i,j);

    push    1
    push    100                 ; 00000064H
    call    ?idifferenz1@@YAHHH@Z           ; idifferenz1
    add esp, 8
```

```
; 28   :
; 29   : k=idifferenz2(i,j);

    push    1
    push    100                 ; 00000064H
    call    ?idifferenz2@@YGHHH@Z           ; idifferenz2

; 30   :
; 31   : k=idifferenz3(i,j);

    mov edx, 1
    mov ecx, 100                ; 00000064H
    call    ?idifferenz3@@YIHHH@Z           ; idifferenz3

; 32   :
; 33   : return 0;

    xor eax, eax

; 34   :
; 35   : }

    ret 0
_main   ENDP
END
```

13.6 Einbindung von Assemblercode in C/C++-Programme

13.6.1 Inline-Assembler in Microsoft Visual C/C++-Programmen (32 Bit)

In C/C++-Programmen, die mit Microsofts Visual C-Compiler übersetzt werden, kann Inline-Assembler eingebunden werden. Dabei hat man bequemen Zugriff auf die Variablen und Funktionen des C/C++-Programms, muss sich allerdings an einige Regeln halten. Diese sind im Folgenden erläutert:

Das _asm-Schlüsselwort

Überall wo ein C/C++-Befehl stehen darf, kann stattdessen auch das Schlüsselwort `_asm` gefolgt von einem Assemblerbefehl stehen. Beispiel:

```
_asm mov eax,0
_asm mov ebx,0x00AAFFFFh
_asm shr ebx,2
```

Nach `_asm` ist auch ein Assemblerblock erlaubt, der von geschweiften Klammern eingeschlossen ist. Die drei obigen Befehle können also auch wie folgt in das C/C++-Programm eingefügt werden:

```
_asm
{
 mov eax,0
 mov ebx,0x00AAFFFFh
 shr ebx,2      ; statt Division durch 4
 }
```

An Inline-Assembler-Zeilen darf nach einem Semikolon ein Assembler-Kommentar angefügt werden.

Zugriff auf C/C++-Symbole

Die Inline-Assembler-Befehle können grundsätzlich auf alle C/C++-Variablen, Funktionen und Sprungmarken zugreifen, die in dem aktuellen Block sichtbar sind. Einschränkungen:

- In jedem Assemblerbefehl kann nur auf ein C/C++-Symbole zugegriffen werden.
- Funktionen auf die zugegriffen wird, müssen vorher deklariert sein.
- Assemblerbefehle können nicht auf C/C++-symbole zugreifen, deren Name in Assemblersprache ein reserviertes Wort sind, z.B. eax, esi, egs, mov, test, aaa, lods usw.
- structure- und union-tags werden in Inline-Assembler nicht erkannt.

Der Umgang mit den Registern

Zunächst einmal kann man zu Beginn einer Assembler-Sequenz nicht annehmen, dass ein Register einen bestimmten Wert hat, der Inhalt der Register ergibt sich aus der Vorbenutzung im normalen Programmablauf. Eine Assemblersequenz oder -funktion darf die Register EAX, EBX, ECX und EDX ohne weiteres ändern. Dagegen sollten die Register EBP, ESP, EDI, ESI, DS, CS und SS nicht verändert bzw. wieder hergestellt werden.

Operatoren zur Größenbestimmung

Mit den Operatoren TYPE, LENGTH und SIZE kann die Größe einer C/C++-Variablen bestimmt werden:
TYPE gibt die Größe einer einzelnen Variablen oder eines Typs zurück
LENGTH gibt die Anzahl der Elemente eines Feldes zurück (Einzelvariable: 1) SIZE gibt die Gesamtgröße einer Variablen oder eines Feldes zurück. Es gilt daher SIZE=TYPE*LENGTH. Beispiel: nach `int feld[5]` ist TYPE feld = 4, LENGTH feld = 5 und SIZE feld = 20.

Zugriff auf C/C++-Variable

Die C/C++-Variablen können direkt über ihren Bezeichner (Namen) angesprochen werden. Es ist aber auch möglich ihr Adresse zu laden und die Variablen dann über den Zeiger zu erreichen. Beispiele:

```
int Anzahl, index, puffer[10];
struct person {
  char *name;
  int alter;
  };
struct person student;

// Zugriff auf einfache Variable
_asm
    {
    ; Direkter Zugriff auf Variablen
    mov eax,Anzahl  ; Laden einer Variablen in ein Register
    shl index,2     ; Bearbeitung einer Variablen

    ; Zugriff mit Adresse und indirekter Adressierung
    lea ebx, Anzahl ; lea = load effective adress
                    ; läd die Adresse (den Offset) von Anzahl nach ebx
    mov [ebx],0     ; Zugriff über den Zeiger (indirekte Adressierung)
    }

// Zugriff auf ein Array nutzen immer die indirekte Adressierung
// Der Index kann fix oder flexibel sein
_asm
    {
    ; Direkter Zugriff auf ein Element eines Arrays
    mov [puffer+4],eax    ; Achtung: kopiert eax in puffer[1]

    ; Zugriff über die Adresse
    lea ebx, puffer       ; Adresse des arrays nach ebx
    mov [ebx+12],20       ; puffer[3]=20

    ; Zugriff mit flexiblem Index
    mov ecx,3             ; C-Feldindex z.B. nach ecx
    mov eax, [puffer+ecx*TYPE puffer]   ; Anfangsadresse + Index*Größe
                                    ; TYPE int = 4, s.o.
    }

// Zugriff auf eine Struktur
_asm
    {
    ; Direkter Zugriff auf Elemente der Struktur
    mov student.alter, 23   ; direkter Zugriff
```

```
        mov esi,student.name    ; Zeiger nach esi laden

        ; Zugriff auf die Struktur über die Adresse
        lea ebx, student        ; Adresse
        mov ecx,[ebx].alter     ; Zugriff
        }
```

Konstanten können wie Assemblerkonstanten geschrieben werden oder wie C-Konstanten, z.B.

```
    _asm
        {
        mov eax, 001FFFFFh  ; Assembler-Schreibweise
        mov eax, 0x1FFFFF   ; C-Schreibweise
        }
```

Sprungbefehle

Als Sprungziele können sowohl Inline-Assembler-Sprungmarken genannt werden, als auch C/C++-Sprungmarken. Alle Sprungmarken können sowohl durch Assemblerbefehl (z.B. jmp) oder durch C/C++-Befehl (goto) angesprungen werden. Groß-/Kleinschreibung muss dabei nur beachtet werden, wenn mit goto eine C/C++-Sprungmarke angesprungen wird. Die Namen von Sprungmarken sollten nicht mit den Namen von C-Bibliotheksfunktionen übereinstimmen, z.b. exit.

Operatoren

Zeichen, die in C/C++ und Assembler Operatoren bezeichnen, wirken in Inline-Assembler als Assembleroperatoren. Beispiel: der Stern (*) bezeichnet keinen Zeiger sondern eine Multiplikation bei der Adressberechnung (Index-Skalierung)

Anlegen von Daten

In Inline-Assembler können keine Daten angelegt werden, die Direktiven DB, DW, DD usw. sind nicht erlaubt.

Aufruf von C/C++-Funktionen

Der Aufruf von C/C++-Funktionen ist möglich, der Assemblercode muss nur vorher die Parameter auf dem Stack hinterlegen. Beispiel:

```
char text[]="Zaehler = %i\n";
int zaehler=100;

_asm
```

```
    {
    mov eax, zaehler
    push eax
    lea eax, text
    push eax
    call printf
    add esp,8
    }
```

C++-Funktionen können nur aufgerufen werden, wenn sie global sind.

Die Rückgabe von Funktionsergebnissen

Der MS Visual C-Compiler (und andere Compiler ebenfalls) legt die Programme so an, dass Funktionsergebnisse möglichst im Register EAX bzw. Teilen davon zurückgegeben werden. Funktionen in Inline-Assembler müssen sich an die gleichen Konventionen halten. Die folgende Tabelle gibt einen Überblick.

Funktionstyp	Rückgaberegister
char	AL
short	AX
int,long	EAX
real, double	Numerikeinheit-Register st(0)
Strukturen bis zu 64 Bit	EDX–EAX
Strukturen größer als 64 Bit	Zeiger auf Speicherbereiche

Ein Beispiel soll einige Zugriffe in Inline-Assembler in MSVC demonstrieren:

```
#include <stdio.h>

// Funktion in Inline-Assembler
int Produkt(int Faktor1, int Faktor2) {
_asm
    {
    mov eax,Faktor2         // zweiter Parameter in eax
    imul eax,Faktor1        // ersten Parameter damit multiplizieren,
                            // Ergebnis bleibt in eax zur Rückgabe
    }
return;                     // Hier wird ein Warning erzeugt, da scheinbar
                            // der Rückgabewert fehlt
}

main() {
char char1;
```

```
short short1;
int int1,int2,int3;
int *p1, *p2;
int **pp;
int x[10];

// Zugriff auf Variable
_asm mov al,'A'
_asm inc al
_asm mov char1,al           // char1 = 'B' = 66

short1=200;
_asm sar word ptr short1,1  // short1 = 100

_asm mov int1,0FFFFFF00h    // int1 = -256d
printf("char1=%c, short1=%i, int1=%i\n", char1, short1, int1);


// Spruenge
_asm
    {
    mov eax,int1
    cmp int2,eax            // int1 <> int2
    je weiter               // Sprung zu C-Sprungmarke
    cmp eax,10
    jl Marke1               // Sprung zu Assembler-Sprungmarke
    mov dword ptr int2,0
    marke1:
    }
weiter:


// Benutzung von Zeigern
_asm lea ebx,int2
_asm mov p1,ebx                 // p1 erhält die Adresse von int2
_asm mov dword ptr [ebx],99;    // *p1 = int2 = 99
_asm mov p2,ebx                 // p2 enthält die gleiche Adresse wie p1, p2=p1
printf("int1=%i\n",int2);

// Zeiger auf Zeiger
int3=25;
p1=&int3;   // p1 enzhält die Adresse von int3
pp=&p1;     // pp enthält Adresse von p1
_asm mov ebx,pp     ; Inhalt von Zeiger pp (Adresse von p1) nach ebx
_asm mov ebx,[ebx]  ; Inhalt von Zeiger p1 (Adresse von int3) nach ebx
_asm mov eax, [ebx] ; Zugriff auf int3,
_asm mov int1,eax       ; Kopie auf int1
printf("int1=%i, int3=%i\n",int1,int3);
```

```
// Zugriff auf Felder, hier Initialisierung eines Feldes mit 0xFFFFFF
_asm
    {
    mov ecx,10
    mov eax,0FFFFFFh
l1: mov dword ptr[x+ecx*4-4],ecx
    loop l1
    }
for (int1=0; int1<10; int1++)
    printf("%3i",x[int1]);

// Aufruf der Inline-Assembler-Funktion
int1=33;
int2=100;
int3=Produkt(int1,int2);
printf("%10i\n",int3);

return 0;
}
/* Programmausgabe:

  char1=B, short1=100, int1=-256
  int1=99
  int1=25, int3=25
  1  2  3  4  5  6  7  8  9 10    3300

*/
```

13.6.2 Inline–Assembler in Borland C-Programmen (16-Bit)

Hier sind beispielhaft die Verhältnisse für den 16-Bit-C-Compiler von Borland angegeben. Das Format einer Inline–Assembleranweisung ist

`asm` *Assemblerbefehl*

Für mehrere aufeinanderfolgende Befehle kann auch wie folgt verfahren werden:

`asm { ` *Assemblerbefehl*
Assemblerbefehl
Assemblerbefehl
`...}`

Dabei gelten für die Assemblerbefehle folgende Regeln:

1. Die Operanden der Befehle dürfen auch Konstanten, Variablen und Labels (Sprungmarken) des umgebenden C–Programms sein.

2. Der Assemblerbefehl wird mit Zeilenvorschub (CR) oder Semikolon (;) abgeschlossen, mit einem Semikolon kann aber *kein* Kommentar eingeleitet werden.
3. Ein Kommentar muß als C–Kommentar bzw. C++ –Kommentar geschrieben werden, also in /∗ . . . ∗/ bzw. //
4. Sprungbefehle in Inline–Assembler dürfen sich nur auf C–Labels beziehen
5. Befehle die nicht verzweigen, dürfen alles *außer* C–Labels verwenden
6. in Inline–Assembler wird keine automatische Größenanpassung beim Zugriff auf Speichervariable gemacht, sie müssen ggf. explizit durch `BYTE PTR, WORD PTR` usw. vorgenommen werden.

```
#include "stdio.h"
#include "conio.h"

/*  Demonstration der Anwendung von Inline-Assembler
in C bzw. C++ Programmen */

main() {

/*  Definition von C-Variablen  */
unsigned char c1,c2;
short k;
char puffer[80]="Morlenstund hat gold im Mund";
char *P;
char far *FP;
char **PP;

printf("\n\n Demo-Programm: Inline-Assembler in C-Programmen\n");

/* Zugriff auf Bytevariable */
c1='A';
asm mov al,c1             // Lesen einer Speichervariablen
asm inc al
asm mov c2,al             // Schreiben einer Speichervariablen
printf("c1=%c  c2=%c\n",c1,c2);

asm inc byte ptr c2       // Bei moeglicher Mehrdeutigkeit
                          // muss die Datenbreite angegeben werden:
printf("c2=%c\n",c2);


/* Zugriff auf Wortvariable */
k=50;
asm mov ax,k
asm shl ax,1
asm mov k,ax
```

```
printf("k=%i\n",k);
asm dec word ptr k        // Angabe der Datenbreite notwendig
printf("k=%i\n",k);

/* Zugriff auf Felder */
printf("%s\n",puffer);    // Vor Bearbeitung
asm lea bx,puffer         // LEA ("Load Effective Adress") laed
                          // die Adresse von "puffer"
asm mov byte ptr [bx+3],'g'  // Ind. Adressierung, Zugriff auf Position 3
                             // (erstes Zeichen liegt an Pos. 0), 1. Textfehler
printf("%s\n",puffer);       //

/* Benutzung von Zeigern */
P=puffer;
asm mov bx,P
asm sub byte ptr [bx+16],'a'-'A'    // dereferenzieren durch indirekte
// Adressierung von "puffer" ; Umwandlung in Grossbuchstaben, 2.Textfehler
printf("%s\n",puffer);

/* Arbeiten mit Zeigern auf Zeiger */
P=puffer;
PP=&P;                    // PP enthaelt Zeiger auf Zeiger auf "puffer"
asm mov bx,PP
asm mov si,[bx]           // dereferenzieren, Zeiger auf "puffer" nach SI
asm mov cl, byte ptr [si]  // nochmal dereferenzieren,
                          // Zugriff auf "puffer" (erstes Zeichen lesen)
asm mov byte ptr c1, cl
printf("c1=%c\n",c1);

/* Arbeiten mit far pointern */
FP=puffer;
asm push ds   // Inhalt von DS aufbewahren
asm lds si,FP     // LDS SI, ("Load Pointer to DS and SI") laedt
                  // die Adresse von "puffer" nach DS:SI
asm lodsb
asm mov c2,al
asm pop ds
printf("c2=%c\n",c2);

getch();
return 0;
}
```

13.6.3 Externe Assemblerprogramme in Borland C-Programmen (16 Bit)

Der folgende Abschnitt beschreibt die Verhältnisse für den 16-Bit-C-Compiler von Borland und den Turbo Assembler.

Einbindung externer Assemblerprogramme

Das Übersetzen und Binden von C/C++– und Assemblermodulen kann mit folgender Kommandozeile ausgeführt werden:

```
BCC CPROG1 CPROG2 ... ASMPROG1.ASM ASMPROG2.ASM ...
```

Dabei muß für die ASM–Dateien die Namenserweiterung `.ASM` angegeben werden. Die erzeugte lauffähige Datei heißt `CPROG1.EXE`.

Alternativ lassen sich gemischte Programme bequem in sog. Projekten verwalten, die in der integrierten Entwicklungsumgebung von Borland C angelegt werden. In Projekten können Dateien folgender Typen eingebunden sein:

`.C/CPP, .ASM, .OBJ, .LIB`.

Dabei ist für den automatisch erfolgenden Aufruf von `TASM` die Option /MX gesetzt, so daß bei allen externen Symbolen in dem Assemblerquelltext zwischen Groß– und Kleinschreibung unterschieden wird.

Regeln für gemeinsame Funktionen und Variablen

Bei Benutzung der vereinfachten Segmentdirektiven ergeben sich folgende Regeln für den Aufruf externer Assemblerprogramme aus C–Programmen:

1. Alle Module müssen im gleichen Speichermodell übersetzt sein, bei den Assemblermodulen kann dabei die Anweisung `.MODEL` benutzt werden.
2. Variablennamen, die in C– und Assemblermodulen vorkommen, müssen in den Assemblermodulen mit einem vorangestellten Unterstrich (Underscore) geschrieben werden [1].
3. Externe Assemblerfunktionen werden in den C–Modulen als `extern` und in C++–Modulen als `extern C` deklariert; in dem Assemblermodul, in dem sie codiert sind, werden sie als `PUBLIC` deklariert.
4. Daten, die in einem C–Modul angelegt sind, können in einem Assemblermodul benutzt werden, wenn sie dort im Datenbereich (nach `.DATA`) als `EXTRN` unter Angabe des Datentyps (s.u.) deklariert sind.
5. Initialisierte Daten werden in Assemblermodulen unter `.DATA` definiert, nicht initialisierte Daten unter `.DATA?`. Sie können in C–Modulen benutzt werden, wenn sie dort als `extern` deklariert sind.

Datentyp von C–Variablen in Assembler

Wenn in einem Assemblerprogramm C–Variablen mit `EXTRN` deklariert werden, muß der Assembler–Datentyp angegeben werden, der dem C–Datentyp dieser Variablen entspricht, z.B:

In einem C–Programm:

[1] Eine Alternative bietet die Deklaration als `EXTRN C`

```
char  c  /* wird in externer Assemblerfunktion benutzt */
.
.
```

und in einer externen Assemblerroutine

```
EXTRN _c:Byte    ; Import aus C--Modul
.
.
       DEC BYTE PTR [_c]
```

Welche Datentypen in Assembler den C–Datentypen entsprechen zeigt die folgende Tabelle

Datentyp in C	Datentyp in Assembler	Länge in Byte
unsigned char	byte	1
char	byte	1
enum	word	2
unsigned short	word	2
short	word	2
unsigned int	word	2
int	word	2
unsigned long	dword	4
long	dword	4
float	dword	4
double	qword	8
long double	tbyte	10
near *	word	2
far *	dword	4

Rückgabe von Funktionsergebnissen

Wenn eine Assemblerfunktion einen Funktionswert zurückgibt, muß sie das in der gleichen Art tun wie eine C–Funktion. Dies erfolgt über Register wie in der folgenden Tabelle aufgeführt ist.

Datentyp in C	Register
unsigned char	`AX`
char	`AX`
enum	`AX`
unsigned short	`AX`
short	`AX`
unsigned int	`AX`
int	`AX`
unsigned long	`DX:AX`
long	`DX:AX`
float	`8087 ST(0)`
double	`8087 ST(0)`
long double	`8087 ST(0)`
near *	`AX`
far *	`DX:AX`

Die Übergabe von Parametern

In C/C++ erfolgt die Übergabe von Parametern bei Funktionsaufrufen über den Stack. Vor dem Aufruf werden die Parameter mit `PUSH` auf dem Stack abgelegt, wobei der letztgenannte Parameter zuerst abgelegt wird. Beim Aufruf der Funktion wird im nächsten Schritt die Rücksprungadresse auf dem Stack abgelegt. Bei Speichermodellen mit max. 64 kB Code ist dies `IP`, bei Speichermodellen mit mehr Code ist es `CS:IP`. Um den Stack zu adressieren wird `BP` benutzt. Da die rufende Funktion ein nach Ausführung der aufgerufenen Funktion unverändertes `BP` erwartet, muß der aktuelle Wert von `BP` selbst auf den Stack gerettet werden, der erste Befehl im Unterprogramm ist also `PUSH BP`. Danach wird `BP` zur Adressierung des Stack vorbereitet, was mit `MOV BP,SP` geschieht. Durch Vermindern von `SP` kann nun Platz für lokale Variable reserviert werden.

Nun werden die eigentlichen Befehle des Unterprogramms ausgeführt, wobei man Parameter wie auch lokale Variable über `BP` adressiert. Nach dem Rücksprung in die rufende Funktion müssen dort die Parameter wieder vom Stack entfernt werden.

Für die Speichermodelle `TINY, SMALL` und `COMPACT` ergibt sich also während der Ausführung eines Unterprogramms folgender Stackaufbau:

Adresse	Inhalt
`BP + 10`	.
`BP + 8`	.
`BP + 6`	zweites Parameterwort
`BP + 4`	erstes Parameterwort
`BP + 2`	Rücksprungadresse
`BP`	geretteter `BP`
`BP - 2`	erstes Wort der lokalen Variablen
`BP - 4`	zweites Wort der lokalen Variablen
`BP - 6`	drittes Wort der lokalen Variablen
`BP - 8`	.
`BP - 10`	.

Für die Speichermodelle `MEDIUM, LARGE` und `HUGE` ergibt sich folgender Stackaufbau:

Adresse	Inhalt
`BP + 12`	.
`BP + 10`	.
`BP + 8`	zweites Parameterwort
`BP + 6`	erstes Parameterwort
`BP + 4`	Rücksprungadresse
`BP + 2`	Rücksprungadresse
`BP`	geretteter `BP`
`BP - 2`	erstes Wort der lokalen Variablen
`BP - 4`	zweites Wort der lokalen Variablen
`BP - 6`	drittes Wort der lokalen Variablen
`BP - 8`	.
`BP - 10`	.

Kapitel 14

Assemblerpraxis

14.1 Der Zeichensatz

Wir beziehen uns hier auf den ASCII-Zeichensatz. Es handelt sich dabei um einen 7-Bit-Zeichensatz, d.h. bei Benutzung von 8-Bit.-Einheiten ist das MSB immer Null. Dieser ist in *Steuerzeichen* und *darstellbare Zeichen* aufgeteilt. Die ersten 32 Zeichen (0–31, 0–1Ah) sind die Steuerzeichen; mit Ihnen kann z.B. die Position des Cursors auf dem Bildschirm gesteuert werden. Einige Steuerzeichen sind unten wiedergegeben.

Dez.	Hex.	Kurzbez.	Bedeutung
7	07	BEL	Klingelzeichen, Piepen
8	08	BS	Backspace: Ein Zeichen nach links löschen
9	09	HT	Horizontaler Tabulator
10	0A	LF	Line Feed: Cursor eine Zeile tiefer stellen
11	0B	VT	Vertikaler Tabulator
12	0C	FF	Formfeed: Neue Seite auf dem Drucker anfangen
13	0D	CR	Carriage Return: Wagenrücklauf, Cursor springt ganz nach links

Tabelle 14.1: Einige Steuerzeichen

Die darstellbaren Zeichen beginnen bei Nummer 32 (20h). Sie sind direkt zur Ausgabe auf dem Bildschirm gedacht. Zur Erleichterung ist die Nummer der Zeichen dezimal und hexadezimal angegeben.

Dez.	Hex.	Zeichen
32	20	
33	21	!
34	22	”
35	23	#
36	24	$
37	25	%
38	26	&
39	27	’
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	3E	.
47	3F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dez.	Hex.	Zeichen
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	ˆ
95	5F	_

Dez.	Hex.	Zeichen
96	60	‘
97	61	a
98	62	b
99	63	b
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	˜
127	7F	⌂

Tabelle 14.2: Die darstellbaren Zeichen des ASCII-Zeichensatzes

Binär	Hexadezimal	Dezimal	Binär	Hexadezimal	Dezimal
0000b	0h	0d	1000b	8h	8d
0001b	1h	1d	1001b	9h	9d
0010b	2h	2d	1010b	Ah	10d
0011b	3h	3d	1011b	Bh	11d
0100b	4h	4d	1100b	Ch	12d
0101b	5h	5d	1101b	Dh	13d
0110b	6h	6d	1110b	Eh	14d
0111b	7h	7d	1111b	Fh	15d

Tabelle 14.3: Die Zahlen von 0 – 15 in binärer, dezimaler und hexadezimaler Darstellung

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
MSB							LSB

Tabelle 14.4: Zählung der Bits in einem Byte

14.1.1 Informationseinheiten

In Mikroprozessorsystemen werden (fast) immer mehrere Bit zu einer Informationseinheit zusammengefasst:

- 4 Bit sind eine *Tetrade* oder ein *Nibble*
- 8 Bit sind ein *Byte*
- die Verarbeitungsbreite des Prozessors umfasst ein *Maschinenwort* oder *Wort*; bei einem Prozessor mit 32-Bit-Verarbeitungsbreite sind also 4 Byte ein Maschinenwort
- Ausgehend vom Maschinenwort wird auch von *Halbworten, Doppelworten* und *Quadworten* (vier Maschinenworte) gesprochen; bei einem 16 Bit-Prozessor umfasst ein Quadwort 64-Bit

Für die Darstellung auf Papier oder Bildschirm wird in der Regel die *Hexadezimale Darstellung* gewählt, dabei stellt jede Hexadezimalziffer 4 Bit dar. Ein Byte kann also immer mit 2 Hexziffern dargestellt werden. Um Verwechslungen zu vermeiden benutzt man Anhänge, die das verwendete Zahlensystem kennzeichnen: *b=binär, d=dezimal, h=hexadezimal*

Das niedrigstwertige Bit innerhalb eines Bytes oder Wortes heißt *least significant bit* (LSB) das höchstwertige heißt *most significant bit* (MSB). Die Nummerierung der Bits innerhalb eines Bytes oder Wortes *beginnt immer beim LSB mit 0* (s.Tab.14.4). Für größere Informationseinheiten gibt es gebräuchliche Abkürzungen die an die Einheitenvorsätze der Naturwissenschaften angelehnt sind, wie z.B. das kiloByte (s.Tab.14.5).

14.2 Die DOS-Kommandozeile - zurück in die Steinzeit

Bei den praktischen Übungen mit Assemblerprogrammen benutzt man evtl. die Kommandozeile unter MS-DOS. Dabei stehen nicht die gewohnten grafischen Windows-Tools zur Verfügung,

2^{10} Byte	=	ein Kilobyte	=	1 kB	=	1024 Byte	=	1024 Byte
2^{20} Byte	=	ein Megabyte	=	1 MB	=	1024 kB	=	1048576 Byte
2^{30} Byte	=	ein Gigabyte	=	1 GB	=	1024 MB	=	1073741824 Byte
2^{40} Byte	=	ein Terabyte	=	1 TB	=	1024 GB	=	1099511627776 Byte

Tabelle 14.5: Bezeichnungen für größere Informationseinheiten

stattdessen werden hier Kommandozeilenbefehle benutzt. Mit diesen kann man aber genausogut (manchmal sogar besser) seine Dateien und Verzeichnisse verwalten. Im Folgenden sind einige Befehle vorgestellt. Wichtig:

- Jede Eingabe muss mit der Return-Taste (Enter, Pfeil mit Haken) abgeschlossen werden
- es werden keine langen Namen unterstützt. Dateinamen dürfen max. 8+3 Zeichen haben, Verzeichnisnamen max. 8 Zeichen.
- Die DOS-Kommandozeile unterscheidet nicht zwischen Groß- und Kleinschreibung.
- In Dateinamen kann ein Stern (*) als sog. Joker eingesetzt werden, dieser steht dann für alle Dateinamen. Beispiel: `*.asm` steht für „alle Dateinamen mit Extension `asm`“

Prompt

Unter Prompt versteht man das Bereitschaftszeichen am Anfang der Zeile. Meist ist der Prompt so eingestellt, dass er das aktuelle Laufwerk und das aktuelle Verzeichnis anzeigt. (Falls nicht, kann er mit dem Kommando `prompt $P$G` so eingestellt werden.) Bsp:

```
C:\ASM\>
```

bedeutet: Das aktuelle Laufwerk ist C: (Festplatte), das aktuelle Verzeichnis ist `\ASM`, der Rechner ist bereit und wartet auf ein Kommando.

Aktuelles Verzeichnis einstellen: CD

Die Verzeichnisse sind strukturiert wie unter Windows. (dort heißen sie Ordner) Um das aktuelle Verzeichnis einzustellen benutzt man den Befehl `CD` (Change Directory). Beispiele:

`CD`	Zeigt das eingestellte aktuelle Verzeichnis an
`CD␣asm`	wechselt ins Unterverzeichnis `asm`
`CD␣..`	wechselt ins nächst höhere Verzeichnis
`CD\`	wechselt ins oberste Verzeichnis (Root-Directory)

Dateien auflisten: DIR

Mit dem Befehl `DIR` (Directory, Verzeichnis) listet man den Inhalt eines Verzeichnisses auf. Dabei sind verschiedene Parameter erlaubt.

Beispiele:

`dir`	Listet die Dateien im aktuellen Verzeichnis und Laufwerk
`dir␣a:`	Listet die Dateien im aktuellen Verzeichnis von Laufwerk `A:`
`dir␣asm`	Listet die Dateien im Verzeichnis `asm`

Dateien kopieren: COPY

Mit dem Befehl `COPY` kann man Dateien kopieren. Die Kopieen können auf anderen Laufwerken, in anderen Verzeichnissen oder in Dateien mit anderem Namen angelegt werden. In dem Befehl folgt nach dem Wort `COPY` zunächst die Quelle der Daten, dann das Ziel. Beispiele:

`copy␣auf1.asm␣a:`	Kopiert Datei `auf1.asm` auf Laufwerk `A:`
`copy␣*.asm␣a:`	Kopiert alle Datei mit Extension `asm` auf Laufwerk `A:`
`copy␣a:auf1.asm␣c:\maschpr`	Kopiert Datei `AUF1.ASM` von `A:` nach `C:\MASCHPR`
`copy␣a:auf1.asm`	Kopiert Datei `AUF1.ASM` von `A:` ins aktuelle Verzeichnis von C:

14.3 Assemblieren, Linken Debuggen

Um aus einem in Assembler geschriebenen Programm eine ausführbare Datei zu erzeugen, muß man den Sourcefile (*Dateiname*.ASM) assemblieren und linken. Das geht z.B. mit dem Turbo Assembler von Borland mit der Kommandozeile

`TASM` *Dateiname*`.ASM`

Der Assembler hat nun einen Object-File (*Dateiname*.OBJ) erzeugt. Dieser wird mit dem Turbo Linker gelinkt (gebunden):

`TLINK` *Dateiname*`.OBJ`

Aus dem Assemblerprogramm ist nun ein ausführbarer *Dateiname*.EXE-File geworden. Er kann nun mit *Dateiname* CR aufgerufen werden.

Beim allen Aufrufen kann die Extension des Dateinamens weggelassen werden, also TASM *Dateiname*, TLINK *Dateiname*.

Um das Programm zu testen, ist ein Debugger sehr nützlich. Der Turbo Debugger (TD) kann vorteilhaft benutzt werden, wenn beim Assemblieren und Linken zusätzliche Informationen eingefügt werden. Dies geschieht durch folgende Optionen:

```
TASM /zi Dateiname
TLINK /v Dateiname
TD Dateiname
```

Zum Editieren der Assemblerprogramme kann ein beliebiger Editor benutzt werden, z.B. EDIT, Ultraedit, Winedt usw. Der Turbo Debugger ist auch ein gutes Lernhilfsmittel. Nach dem Aufruf steht er vor der ersten ausführbaren Zeile. Mit ALT-V-R kann das Register-Fenster geöffnet werden, mit F8 kann ein Einzelschritt ausgeführt werden. dabei kann man alle Register und Flags beobachten. Ein Programmneustart erfolgt mit CTRL-F2, ein Breakpoint kann mit F2 gesetzt werden.

Zu jedem Fenster gibt es ein „lokales Menü“, das mit ALT-F10 oder der rechten Maustaste aufgerufen wird. Die Hilfetaste (F1) gibt Hilfstexte aus. Im „CPU-Fenster“ hat man eine besonders maschinennahe Sicht auf das Programm: Man sieht den erzeugten Maschinencode, die Register, die Flags einen Ausschnitt aus dem Datenspeicher und einen Ausschnitt aus dem Stack. Auch hier gibt es nützliche Menüs.

14.4 Ein Rahmenprogramm

Bei Verwendung des Borland Assemblers und der sog. vereinfachten Segmentanweisungen kann z.B. folgender Rahmen für vollständige Programme benutzt werden:

```
;
;  Titel des Programmes: . . . . . .
;
   DOSSEG
   .MODEL Small  ; moegl. sind Tiny,Small,Medium,Compact,Large,Huge
   .STACK 256    ; Stackgroesse 256 Byte, max moeglich sind 64 kB

; EQU und = Direktiven hier einfuegen

   .DATA

; Reservierung von Speicherplatz und Zuordnung von Namen fuer Variable
bytevariable1        DB ?           ; nicht initialisiertes Byte
Wortvariable1        DW 0           ; initialisierte Wortvariable
Feld1                DW 10 DUP(0)   ; 10 Worte, mit dem Wert Null initialisiert

; Initialisierter String
titel       DB 'Programmtitel: MUSTER.ASM ',13,10,'$'

    .CODE

Programmstart:           ; Label haben einen Doppelpunkt am Ende
   mov  ax,@data             ; Uebergabe der Adresse des Datensegments
                             ; zur Laufzeit
   mov  ds,ax                ; DS zeigt nun auf das Datensegment
```

```
   mov  ah,9                  ; DOS-Funktion "print string"
   mov  dx,OFFSET titel       ; Adresse von "titel"
   int  21h                   ; Ueberschrift ausgeben
   ...
   ; eigentliches Programm
   ...

; Programmende, Kontrolle explizit an DOS zurueckgeben
;
EXIT:
   mov  ah,04Ch          ; ah=04C : DOS-Funktion "Terminate the program"
   mov  al,0             ; DOS-Return-Code 0
   int  21h              ; Interrupt 21h : Aufruf von DOS

;  Unterprogramme z.B. hier einfuegen,
;  auch mit Include-Anweisungen


   END Programmstart
```

Kapitel 15

Lösungen zu den Testfragen

Lösungen zu den Fragen aus Abschnitt Speicherbenutzung

(Abschnitt 2.5)

1. Überlegen Sie ob die folgenden Befehle korrekt sind:

```
.DATA
Zaehler1 DB ?
Zaehler2 DB 0
Endechar DB ?
Startchar DB 'A'
Pixelx DW ?
Pixely DW 01FFh
Schluessel DD 1200h
.CODE
mov Zaehler1, 100h   ; Konstante zu gross für 8-Bit-Variable
mov Zaehler2, ax     ; Register hat 16 Bit, Speichervariable hat 8 Bit

mov ah,2
mov dx, Startchar    ; funktioniert, setzt allerdings unnötigerweise dh=0
int 21h

movzx Endechar,'Q'   ; movzx wird nicht gebraucht bei zwei 8-Bit-Operanden
mov edx,Startchar    ; statt mov muss movzx benutzt werden
xchg Pixely,cx       ; o.k.

mov schluessel,ebp   ; o.k.
mov Pixelx,Pixely    ; geht nicht, da zwei Speicheroperanden in einem Befehl
```

2. Überlegen Sie welche der folgenden Befehle zu Fehlermeldungen, Warnungen oder Laufzeitfehlern führen:

```
.DATA
Nummer DB 25 DUP (0)
zahl DW 0
.CODE
.386
mov [Nummer+cx],al    ; mit cx kann nicht adressiert werden

mov [Nummer+ecx],al   ; o.k., da beliebige 32-Bit Register erlaubt

mov al,[Nummer+bl]    ; mit 8-Bit-Registern kann nicht adressiert werden

mov [bx+bp+10],0    ; nicht erlaubt, zwei Basisregister

mov [si+di+1],10h   ; nicht erlaubt, zwei Indexregister

mov bx, offset zahl
mov cl, [Nummer+bx]   ; Laufzeitfehler, Adresse liegt hinter dem Feld Nummer

mov cl, Nummer        ; o.k., überträgt das erste Byte von Nummer

inc [bx]
```

3. Speicher: 66 11 07 22 08 33 09 00 13 00 12
   ax=1166h, cx=2207h, edx=09330822h, esi=00130009h

Lösungen der Fragen zu den Transportbefehlen

(Abschnitt 3.5)

1.
```
 1:  mov al,50h          ; ok
 2:  mov al,100h         ; Fehler, Maximalwert bei 8-Bit-Registern ist FFh
 3:  mov 22,bh           ; Fehler, Ziel kann keine Konstante sein
 4:  mov cx,70000o       ; ok
 5:  mov cx,70000        ; Fehler, Maximalwert 16-Bit-Reg.: FFFFh=65535d
 6:  mov bx, 10001111000000b    ; ok
 7:  mov eax,177FFA001h         ; Fehler, Maximalwert 32-Bit-Reg.: FFFFFFFFh
 8:  mov edx, 02A4h      ; ok, Konstante wird erweitert zu 000002A4h
 9:  xchg cx,10h         ; Fehler, Austausch mit Konstante nicht möglich
10:  mov eax,-1          ; ok, -1 wird vom Assembler im Zweierkomplement eingesetzt
11:  mov eax,edi         ;ok
12:  mov ah,bl           ;ok
13:  mov bx,bl           ;Fehler, Operanden versch. Bitbreite: BX: 16 Bit, BL: 8 Bit
14:  xchg eax,bp         ;Fehler, Operanden versch. Bitbreite: EAX: 32 Bit, BP: 16 Bit
15:  xchg dx,dx          ;ok, bewirkt aber nichts!
16:  mov dl,di           ;Fehler, Operanden versch. Bitbreite: DL: 8 Bit, DI: 16 Bit
17:  mov bp,bh           ;Fehler, Operanden versch. Bitbreite: BP: 16 Bit, BH: 8 Bit
```

```
18:  xchg edi,dl          ;Fehler, Operanden versch. Bitbreite: EDI: 32 Bit, DL: 8 Bit
19:  mov esi,dx           ;Fehler, Operanden versch. Bitbreite: ESI: 32 Bit, DX: 16 Bit
20:  xchg esi,ebx         ;ok
21:  xchg ch,cx           ;Fehler, Operanden versch. Bitbreite: CH: 8 Bit, CX: 16 Bit
22:  mov ch,cl            ;ok
```

2. EAX=12345678h

3. Zeilen 1,2: Diese beiden Befehle können ersetzt werden durch mov ax,100h
   Zeilen 3,4: Die beiden Befehle können ersetzt werden durch mov ebx,2800h
   Zeilen 5,6: Diese beiden Befehle können ersetzt werden durch movzx eax,dl
   Zeile 7: Entspricht No Operation (NOP), ändert kein Register u. kein Flag
   Zeilen 8–10: Soll eine Vertauschung von DI und SI bewirken, besser xchg di,si

4.
```
mov DX,BX   ; Inhalt von BX parken (retten)
            ; geht nur, wenn DX frei
mov BX,AX
mov AX,CX
mov CX,DX
```

   b)

```
xchg ax,bx
xchg ax,cx
```

5.
```
; a)
    mov si,ax
    shr eax,16
    mov di,ax

; b)
    mov ax,dx
    shl eax,16
    mov ax,cx

; c)
    mov dl,cl
    mov dh,ch

    ; oder besser und kürzer
    mov dx,cx
```

6. `movzx eax,ax`

Lösungen der Fragen zu den Betriebssystemaufrufen

(Abschnitt 5.4)

Abschnitt 1: Nummer des Funktionsaufrufes wurde nicht in AH hinterlegt.
Abschnitt 2: int 21 dezimal statt 21h wird aufgerufen, Achtung tückischer Fehler!
Abschnitt 3: Aufruf korrekt aber sinnlos: Die Ergebnisse in DX und CX werden überschrieben.
Abschnitt 4: Aufruf korrekt, führt aber zur unbeabsichtigten Ausgabe von Zeichen, da die Zeichenkette nicht mit $-Zeichen begrenzt ist.
Abschnitt 5: Vorbereitung korrekt aber Betriebssystemaufruf (hier Int 21h) fehlt.

Lösungen der Testfragen zu den Bitbefehlen

(Abschnitt 6.4)

1. AX=1214h, BX=5335h, CX=FFFFh, DX=ED10h

2. al=54h, bl=CAh, cl=56h, dl=55h

3.
```
and ax,1111111111011110b    ; oder and ax,0FFDEh\\
or  ax,0000000000001010b    ; oder or ax,0Ah\\
xor ax,0000000010000100b    ; oder xor ax,84h\\
```

4.
```
mov si,bx   ; Kopie von bx anlegen
shl bx,4    ; bx=bx*16
add bx,si   ; bx=bx*17
mov di,cx   ; Kopie von cx anlegen
shl cx,3    ; cx=cx*8
add cx,di   ; cx=cx*9
add bx,cx   ; bx=17*bx + 9*cx
mov ax,bx   ; Ergebnis nach ax
```

5.
```
; Loesung1:\\
    not bx      ; invertieren\\
    shr bx,7    ; shift right bx,7: Bit 7 ist jetzt LSB\\
                ; es geht auch: rol bl,1\\
    and bx,1    ; LSB stehen lassen, restliche Bits 0 setzen\\
    add ax,bx   ; addieren\\


; Loesung2:\\
    shl bx,8    ; shift left bx,8: Bit 7 ist jetzt MSB\\
    shr bx,15   ; Bit 7 ist jetzt LSB, andere Bits sind 0\\
    xor bx,1\\
    add ax,bx   ; addieren\\


; Loesung3:
    shl bx,9    ; shift left bx,9: Bit 7 jetzt im Carryflag
    mov bx,0    ; Bit 7 ist jetzt LSB, andere Bits sind 0
    rcl bx,1    ; rotate through carry left,CF in LSB von BX
    add ax,bx   ; addieren
```

Lösungen zu den Fragen aus Abschnitt Sprungbefehle

(Abschnitt 7.5)

1. Ergänzen Sie in dem folgenden Programmstück die fehlenden Befehle oder Operanden! (???)

```
; Ausgabe der Ziffern von '9' abwärts bis '0'
    mov dl, '9'     ;<=== (oder z.B. mov dl,39h)
Schleifenstart:
    mov ah,2        ;DOS-Funktion Zeichenausgabe
    int 21h
    dec dl          ;<===
    cmp dl,'0'      ;<=== (oder z.B. mov dl,30h)
    jae Schleifenstart
```

2. AX = 10h

3. Finden Sie die Fehler in dem folgenden Programmstück!

```
; Belegen eines Wortfeldes mit dem Wert 0
.DATA
    Feld DW 20 DUP(?)
.CODE
    mov bx,1        ; Falsch! Richtig ist: mov bx,0
Schleifenstart:
    mov [Feld+bx],0
    inc bx          ; Falsch! Richtig ist: add bx,2
    cmp bx,20       ; Falsch! Richtig ist cmp bx,40
    je Schleifenstart   ; Falsch! Richtig ist: jne oder jb Schleifenstart
```

Lösungen zu den Fragen zu arithmetischen Befehlen

(Abschnitt 8.7)

1. Alle Befehle sind fehlerhaft!

```
    add ax          ; 2.Operand fehlt
    adc bx,ax,cf    ; ein Operand zuviel, CF als Operand nie erlaubt
    mul eax,ebx     ; MUL hat nur einen Operanden
    mul 80h         ; Direktoperand nur bei IMUL erlaubt
    imul ax,bx,cx   ; IMUL hat als dritten Operanden nur Direktwerte
    idiv edx,eax    ; IDIV hat nur einen Operanden
```

2. AX=2000h, BX=0300h, CX=FF90h (-70), DX=0050h, DI=0100h

3. AX=80h, CX=100h

4. AX=0550h, DX=0000h

5. ```
   ;Berechnung von 123456h / 11h
       mov eax,123456h
       mov ebx,11h
       div ebx
   ```

   Dieser Programmabschnitt kann leicht zu einem Divisionsfehler führen, da EDX nicht (mit 0) vorbesetzt ist.

Lösungen zu den Fragen aus Abschnitt Stack

(Abschnitt 9.4)

1. Nur der zweite: Auf dem Stack gibt es keine 8-Bit-Operationen.

2. ax=10, bx=9, cx=8

3. Die Schleife mit dem PUSH-Befehl wird 10 mal ausgeführt, die Schleife mit dem POP-Befehl 11 mal. Der Stack ist also nicht ausbalanciert.
   ```

Kapitel 16

Assemblerbefehle nach Gruppen

16.1 Allgemeines

In diesem Kapitel sind die wichtigsten Befehle nach Gruppen zusammengefaßt und beschrieben. Ausgenommen sind z.B. die Befehle der Gleitkommaeinheit und der MMX-Einheit. Eine vollständige Befehlsreferenz findet man z.B. in den Datenblättern der Prozessorhersteller intel und AMD. Zur Erleichterung des Lesens wird dabei folgende einfache Typographie eingehalten:

- Bezeichnungen, die wörtlich übernommen werden müssen, wie z.B. Befehlsnamen, sind in `Schreibmaschinenschrift` geschrieben.
- abstrakte Bezeichnungen, die noch durch konkrete Angaben ersetzt werden müssen, sind *kursiv* geschrieben.
- Alternativen sind durch einen senkrechten Strich (|) getrennt.
- Optionale Teile sind in eckige Klammern gesetzt([ ])

Bei der Beschreibung jedes Befehls ist ein einheitliches Format eingehalten: In einer Kopfzeile ist das Mnemonic des Befehls, das vollständige engl. Befehlswort und dessen Übersetzung angegeben, z.B.

CMP	Compare Vergleichen

In den beiden nächsten Zeilen sind die Syntax des Befehls und die erlaubten Operanden beschrieben, z.B. **Syntax** `CMP` *Operand1* , *Operand2*

Operand1: *reg8/16/32/mem8/16/32*
Operand2: *reg8/16/32/mem8/16/32*

Dabei steht *reg8/16/32* für Registeroperanden mit 8,16 oder 32 Bit, wie z.B. *AL* bzw. *AX*, und *mem8/16/32* für Speicheroperanden mit 8,16 oder 32 Bit. Manche Befehle erlauben einen *Direktoperanden*, d.h. eine Zahl im Binär–, Oktal–, Dezimal– oder Hexadezimalformat.

In der nächsten Zeile werden die Flags des Prozessorstatusworts aufgeführt, die von dem Befehl verändert werden. In unserem Beispiel sind dies

Flags O S Z A P C

Danach folgt eine knappe Beschreibung der Funktion des Befehls, im Beispiel:

Beschreibung Operand2 wird von Operand1 subtrahiert, die Flags werden wie bei `SUB` gesetzt aber das Ergebnis wird weggeworfen.

Die Auswertung der gesetzten Flags erfolgt meist durch einen direkt nachfolgenden bedingten Sprungbefehl. `CMP` arbeitet für vorzeichenbehaftete und vorzeichenlose Zahlen korrekt. Es können nicht beide Operanden Speicheroperanden sein. Siehe auch → `SUB`

Durch das Pfeilsymbol → (=Verweis) wird auf einen in diesem Zusammenhang interessanten Befehl verwiesen, der ebenfalls in dieser Kurzreferenz beschrieben ist. Abschließend werden zu dem Befehl ein oder mehrere kurze Beispiele gegeben:

Beispiele

```
        ; Abbruch einer Zählschleife mit CMP
        MOV CX,0            ; CX = 0
L1:     CALL Unterprog      ; Schleifenrumpf
        INC CX
        CMP CX,10           ; CX=10 ?
        JNE L1              ; Wenn nicht, Schleife fortsetzen
```

Die Zeile `;---` trennt Beispiele voneinander. Wird in der Beschreibung des Befehls auf Bits Bezug genommen, so ist zu beachten, daß das niederwertigste Bit die Nr. 0 hat.

16.1.1 Das Format einer Assembler–Zeile

Das Format einer Assemblerzeile ist:

[Label] Befehl/Anweisung [Operanden] [Kommentar]

Ebenfalls erlaubt sind reine Kommentarzeilen (Zeilen die mit ; beginnen) sowie Leerzeilen.

16.2 Transportbefehle

MOV	Move Bewegen

Syntax `MOV` *Ziel*, *Quelle*

Ziel: reg8/16/32/mem8/16/32
Quelle: reg8/16/32/mem8/16/32|Direktoperand

Flags —

Beschreibung Allgemeiner und häufig verwendeter Transportbefehl, kopiert den Quelloperanden in den Zieloperanden; der Quelloperand bleibt unverändert.
Einschränkungen:

- Beide Operanden müssen gleiche Bitbreite haben
- Es können nicht beide Operanden Speicheroperanden sein.
- Es können nicht beide Operanden Segmentregister sein.
- Direktoperanden können nicht in Segmentregister geschrieben werden

Für wiederholten Datentransport von|zu Speicherplätzen kommen auch die Befehle → `MOVS`, → `LODS` und → `STOS` in Frage.

Beispiele

```
MOV AX,16
;---
MOV AX,DS            ; Umweg über AX notwendig
MOV ES,AX            ; um DS nach ES zu kopieren
;---
MOV CH,CL            ; Anwendung auf 8 Bit-Register
;---
MOV BX,[BP+4]        ; mit Speicheroperand
```

XCHG — Exchange / Austauschen

Syntax `XCHG` *Operand1* , *Operand2*
Operand1: *reg8/16/32/mem8/16/32*
Operand2: *reg8/16/32/mem8/16/32*

Flags —

Beschreibung Datenaustausch zwischen zwei Registern oder Register und Speicher. Beide Operanden müssen gleiche Bitbreite haben und wenigstens einer von beiden muß ein Registeroperand sein.

Beispiele

```
XCHG CH,CL
;---
XCHG AX,[BX+DI+1]
```

MOVZX/MOVSX — Move with Zero-/Sign Extension / Bewegen mit Null-/Vorzeichenerweiterung

Syntax `movzx/movsx` *Operand1* , *Operand2*
Operand1: *reg16/32/mem16/32*
Operand2: *reg8/16/mem8/16*

Flags —

Beschreibung Datentransport von kleineren in größere Register. Die Daten werden immer auf die niederwertigen Bits geschrieben. MOVZX füllt dabei die frei bleibenden höherwertigen Bits mit Nullen auf. MOVSX füllt die frei bleibenden höherwertigen Bits vorzeichenrichtig auf, d.h. je nach Vorzeichen mit 0 oder 1.

Beispiele
```
MOVZX EAX,DL
;---
MOVSX DX,CL
```

SETcc	SET if cc Bedingtes Setzen

Syntax `SETcc` *Operand*
Operand: *reg8/mem8*

Flags —

Beschreibung Schreibt in den Operanden (=Ziel) eine 1 als 8-Bit-Wert, wenn die mit xx beschriebene Bedingung erfüllt ist und schreibt eine 0 wenn nicht. Die durch cc (Conditions) angedeuteteten möglichen Bedingungen sind die gleichen wie beim bedingten Sprungbefehl Jcc.

Beispiele
```
SETZ CL             ; 01h in cl, wenn Zeroflag gesetzt, sonst 00h in cl
SETGE var8          ; 01h in var8, wenn Größer/gleich-Bedingungerfüllt
                    , sonst eine 00h
```

→ `jne usw.`

16.3 Logische Befehle

NOT	NOT Negation

Syntax `NOT` *Operand*
Operand: *reg8/16/32/mem8/16/32*

Flags —

Beschreibung Führt eine bitweise Negation des Operanden aus, d.h. 0 → 1 und 1 → 0.

Beispiele

```
MOV [mask],11110011b
NOT [mask]          ;mask hat jetzt den Wert 00001100b d.h.12
```

AND|OR|XOR

logical AND|OR|exclusive OR
logisches UND,ODER,exklusives ODER

Syntax `AND|OR|XOR` *Zieloperand*, *Operand2*
Zieloperand: *reg8/16/32/mem8/16/32*
Operand2: *reg8/16/32/mem8/16/32|Direktoperand*

Flags O S Z P C

Hiervon werden OF und CF immer gelöscht!

Beschreibung Diese Befehle führen bitweise logische Operationen zwischen *Zieloperand* und *Operand2* aus, das Ergebnis wird im *Zieloperand* abgelegt. Die Bitoperationen sind

- AND: logisches UND, Ergebnisbit=1 wenn beide Operandenbits=1, sonst Ergebnisbit=0
- OR: logisches ODER, Ergebnisbit=1 wenn mindestens eines der beiden Operandenbits=1, sonst Ergebnisbit=0
- XOR: logisches exklusives ODER, Ergebnisbit=1 wenn die Summe der beiden Operandenbits=1 ist, sonst Ergebnisbit=0

Beispiele

```
AND AL,01111111b   ;Im Register AL Bit 7 löschen
;---
OR [Maske],00000100b   ;In Variable ''Maske'' Bit2 = 1 setzen
;---
XOR [Maske],0F0h   ;high nibble invertieren, low nibble bleibt
```

TEST

Test
Testen

Syntax `TEST` *Operand1*, *Operand2*
Operand1: *reg8/16/32/mem8/16/32*
Operand2: *reg8/16/32/mem8/16/32|Direktoperand*

Flags O S Z P C

Hiervon werden OF und CF immer gelöscht!

Beschreibung Dieser Befehl unterscheidet sich von AND nur dadurch, daß *Operand1* nicht verändert wird.

Beispiele

```
TEST AL,00000010b    ;feststellen, ob im Register AL
                     ; das Bit 1 gesetzt ist, dann ist ZF=0
```

16.4 Schiebe– und Rotationsbefehle

SHR|SHL	Shift right|Shift left Schieben nach rechts|Schieben nach links

Syntax `SHR|SHL` *Operand*, *Anzahl*
Operand: *reg8/16/32/mem8/16/32*
Anzahl: `1|CL`

Flags O S Z P C

Beschreibung Schiebt ("Shiftet´) den Operanden um eine oder mehrere Stellen bitweise nach rechts|links. Beschreibung s. → `SAR, SAL`

Beispiele `s. SAR,SAL`

SAR|SAL	Shift arithmetic right|left Arithmetisches Schieben nach rechts|links

Syntax `SAR|SAL` *Operand*, *Anzahl*
Operand: *reg8/16/32/mem8/16/32*
Anzahl: `1|CL`

Flags O S Z P C

Beschreibung Schiebt ("Shiftet´) den Operanden um eine oder mehrere Stellen bitweise nach rechts|links.

`SAL` ist identisch mit `SHL`. Der Befehl überträgt das MSB ins `CF`, das freiwerdende LSB wird mit 0 besetzt.

`SAR` unterscheidet sich von `SHR`. Bei `SHR` wird das LSB ins `CF` übertragen und MSB wird mit 0 besetzt. Bei `SAR` wird MSB unverändert belassen *und* auf die benachbarte Stelle kopiert. LSB wird ins `CF` übertragen.

Als Anzahl kann entweder `1` oder `CL` angegeben werden. Im ersten Fall wird einmal um ein Bit geschoben, im zweiten Fall so oft wie der Inhalt von `CL` vorgibt. `OF` ist im zweiten Fall undefiniert.

`SHR` und `SHL` können verwendet werden um an einer *vorzeichenlosen* Zahl eine Multiplikation bzw. Division mit|durch $2, 4, 8 \ldots$ durchzuführen. `SAR` und `SAL` können verwendet werden um an einer *vorzeichenbehafteten* Zahl eine Multiplikation bzw. Division mit|durch $2, 4, 8 \ldots$ durchzuführen. S.auch → `ROR, ROL, RCR, RCL`

Beispiele

```
SHL BX,1              ; BX wird mit 2 multipliziert,
                      ; gleichwertig: SAL BX,1
;---
MOV CL,4
```

```
SHR [spalte],CL      ; ''spalte'' vorzeichenlos durch 16 teilen
                     ; Divisionsrest ist unbehandelt
;---
SAR [differenz],1
; ''differenz'' wird durch 2 geteilt, Vorzeichen wird
; korrekt behandelt, Divisionsrest ist im CF
```

ROR|ROL

Rotate right|Rotate left
Rotieren nach rechts|Rotieren nach links

Syntax ROR|ROL *Operand*, *Anzahl*

Operand: *reg8/16/32/mem8/16/32*
Anzahl: 1|CL

Flags | O | | | | | | | C |

Beschreibung Rotiert den Operanden um eine oder mehrere Stellen bitweise nach rechts|links. Das herausfallende Bit wird ins CF *und* auf den freiwerdenden Platz übertragen. Durch ROR wird also LSB nach MSB und (ins CF) übertragen, bei ROL ist es umgekehrt.

Als Anzahl kann entweder 1 oder CL angegeben werden. Im ersten Fall wird einmal um ein Bit rotiert, im zweiten Fall so oft wie der Inhalt von CL vorgibt. OF ist im zweiten Fall undefiniert. S.auch → SHR, SHL, SAR, SAL, RCR, RCL

Beispiele

```
MOV CL,8
ROL AX,CL            ; Gleichwertig mit XCHG AH,AL
```

RCR|RCL

Rotate through Carry right|left
Rotieren durch Carry rechts|links

Syntax RCR|RCL *Operand*, *Anzahl*

Operand: *reg8/16/32/mem8/16/32*
Anzahl: 1|CL

Flags | O | | | | | | | C |

Beschreibung Schiebt ("Shiftet") den Operanden um eine oder mehrere Stellen bitweise nach rechts|links. Durch RCR wird das CF auf das MSB und das LSB ins CF übertragen. Im Unterschied zu ROR|ROL wird also das CF als Teil der rotierenden Einheit betrachtet. Durch RCL wird das MSB ins CF und das CF ins LSB übertragen.

Als Anzahl kann entweder 1 oder CL angegeben werden. Im ersten Fall wird einmal um ein Bit rotiert, im zweiten Fall so oft wie der Inhalt von CL vorgibt. OF ist im zweiten Fall undefiniert. S.auch → ROR, ROL, SHR, SHL, SAR, SAL

Beispiele
```
MOV CL,5
ROL [var1],CL        ; rotiert ''var1'' um 5 Bit
```

16.5 Einzelbit-Befehle

BSF|BSR	Bit Search Forward|Bit Search Reverse Bitsuche vorwärts|Bitsuche rückwärts

Syntax `BSF|BSR` *Operand1* , *Operand2*

Operand1: *reg16/32*
Operand2: *reg16/32/mem16/32*

Flags Z

Beschreibung Durchsucht den ersten Operanden nach dem ersten auftretenden 1-Bit. `BSF` durchsucht vom niedrigstwertigen Bit (LSB) an, `BSR` beginnt beim höchstwertigen Bit (MSB). Die gefundene Bitposition wird im zweiten Operanden abgelegt.

Beispiele
```
BSF EAX,ECX
```

BT|BTS|BTR|BTC	Bit Test|Bit Test and Set |Bit Test and Reset|Bit Test and Complement Bit Testen|Bit testen und setzen|Bit testen und löschen|Bit testen und invertieren

Syntax `BT|BTS|BTR|BTC` *Operand1* , *Operand2*

Operand1: *reg16/32/mem16/32*
Operand2: *reg16/32/Direktoperand*

Flags C

Beschreibung `BT` überträgt ein Bit aus dem ersten Operanden in das Carryflag. Die Bitnummer wird im zweiten Operanden angegeben. `BTS` setzt anschließend dieses Bit im ersten Operanden auf 1, BTR setzt es auf 0 und BTC komplementiert (invertiert) es.

Beispiele
```
BT EAX,5
BTS AX,CX
```

16.6 Arithmetische Befehle

NEG	Negation Negation

Syntax `NEG` *Operand*

Operand: *reg8/16/32/mem8/16/32*

Flags O S Z A P C

Beschreibung Negiert den Operanden im Zweierkomplement, d.h. wechselt dessen Vorzeichen.

Beispiele

```
MOV AX,5000
NEG AX              ; Inhalt von AX ist jetzt -5000
```

INC|DEC	Increment|Decrement Inkrement|Dekrement

Syntax `INC|DEC` *Operand*

Operand: *reg8/16/32/mem8/16/32*

Flags O S Z A P

Beschreibung `INC` erhöht den Operanden um 1, `DEC` erniedrigt den Operanden um 1.

Beispiele

```
      MOV CX,80
l1:   CALL Unterprog
      DEC CX              ; Schleifenzählvariable herabzählen
      JNZ l1              ; bedingter Sprung
```

ADD|ADC	Addition|Addition with Carry Addition|Addition mit Übertrag

Syntax `ADD|ADC` *Operand1* , *Operand2*

Operand1: *reg8/16/32/mem8/16/32*
Operand2: *reg8/16/32/mem8/16/32|Direktoperand*

Flags O S Z A P C

Beschreibung `ADD`: Operand1 wird zu Operand2 addiert, das Ergebnis wird in Operand1 abgelegt.

`ADC`: Operand1 und der Inhalt des Carryflags werden zu Operand2 addiert, das Ergebnis wird in Operand1 abgelegt.

Die Befehle arbeiten für vorzeichenbehaftete und vorzeichenlose Zahlen korrekt. Es können nicht beide Operanden Speicheroperanden sein. Die Kombination aus `ADD` und `ADC` ermöglicht die Addition beliebig großer Zahlen. Siehe auch → `SUB, SBB`

Beispiele

```
; Es soll a = b + c mit 16 Bit-Zahlen berechnet werden
MOV AX,[b]            ; AX = b
ADD AX,[c]            ; AX = b+c
MOV [a],AX            ; a = b+c
;---
.DATA
dvar1 dd              ;doubleword (32 Bit)
dvar2 dd              ; ''
.CODE
MOV AX,word ptr [dvar1]   ;low word von dvar1 nach AX
ADD word ptr [dvar2],AX   ;zu low word von dvar2 addieren
MOV AX,word ptr [dvar1+2]   ;high word von dvar1 nach AX
ADC word ptr [dvar2+2],AX   ;zu high word von dvar2 addieren
                      ;dabei Übertrag (Carry) addieren
```

SUB|SBB

Subtraction|Subtraction with Borrow
Subtraktion mit Borgen

Syntax

`SUB`|`SBB` *Operand1* , *Operand2*

Operand1: *reg8/16/32/mem8/16/32*
Operand2: *reg8/16/32/mem8/16/32*|*Direktoperand*

Flags

O	S	Z	A	P	C

Beschreibung

`SUB`: Operand2 wird von Operand1 subtrahiert, das Ergebnis wird in Operand1 abgelegt.

`SBB`: Operand2 und der Inhalt des Carryflags werden von Operand1 subtrahiert, das Ergebnis wird in Operand1 abgelegt.

Die Befehle arbeiten für vorzeichenbehaftete und vorzeichenlose Zahlen korrekt. Es können nicht beide Operanden Speicheroperanden sein. Die Kombination aus `SUB` und `SBB` ermöglicht die Subtraktion beliebig großer Zahlen. Siehe auch → `ADD, ADC`

Beispiele

```
; Es soll a = b - c mit 16 Bit-Zahlen berechnet werden
MOV AX,[b]            ; AX = b
SUB AX,[c]            ; AX = b-c
MOV [a],AX            ; a = b-c
;---
.DATA
dvar1 dd              ;doubleword (32 Bit)
dvar2 dd              ; ''
.CODE
                      ; 32 Bit Subtraktion
MOV AX,word ptr [dvar1]   ;low Word von dvar1 nach AX
SUB word ptr [dvar2],AX   ;von low Word von dvar2 subtrahieren
MOV AX,word ptr [dvar1+2]   ;high Word von dvar1 nach AX
SBB word ptr [dvar2+2],AX   ;von high Word von dvar2 subtrahieren
;dabei Übertrag (Carry) subtrahieren
```

CMP — Compare / Vergleichen

Syntax `CMP` *Operand1*, *Operand2*

Operand1: *reg8/16/32/mem8/16/32*
Operand2: *reg8/16/32/mem8/16/32|Direktoperand*

Flags O S Z A P C

Beschreibung Operand2 wird von Operand1 subtrahiert, die Flags werden wie bei `SUB` gesetzt aber Operand 1 bleibt unverändert, d.h. das Ergebnis wird "weggeworfen".

Die Auswertung der gesetzten Flags erfolgt meist durch einen direkt nachfolgenden bedingten Sprungbefehl. `CMP` arbeitet für vorzeichenbehaftete und vorzeichenlose Zahlen korrekt. Es können nicht beide Operanden Speicheroperanden sein. Siehe auch → `SUB`

Beispiele

```
        ; Abbruch einer Zählschleife mit CMP
        MOV CX,0            ; AX = b
L1:     CALL Unterprog      ; Schleifenrumpf
        INC CX
        CMP CX,10           ; CX=10 ?
        JNE L1              ; Wenn nicht, Schleife fortsetzen
```

MUL — Multiplication / Multiplikation

Syntax `MUL` *Multiplikator*

Multiplikator: *reg8/16/32/mem8/16/32*

Flags O C

Beschreibung `MUL` führt eine Multiplikation vorzeichenloser Zahlen durch. Im Befehl ist als Operand explizit nur der Multiplikator genannt, der Multiplikand ist immer `AL` bzw. `AX`.

Je nach Bitbreite des Multiplikators wird eine Byte–, Wort– oder Doppelwort–Multiplikation durchgeführt. Bei der Byte–Multiplikation ist der Multiplikand `AL` und das Ergebnis wird in `AX` abgelegt. Bei der Wort–Multiplikation ist `AX` der Operand und das Ergebnis wird in `DX-AX` abgelegt, wobei `AX` das niederwertige Wort enthält. Bei der Doppelwort-Multiplikation ist `EAX` der Multiplikand und das Ergebnis wird in `EDX-EAX` abgelegt. Da ein Overflow nicht möglich ist, werden die Flags `CF` und `OF` dann gesetzt, wenn das Ergebnis die Bitbreite der Quelloperanden überschreitet.

Siehe auch → `DIV, IDIV`

Beispiele

```
; Multiplikation zweier 8 Bit-Speicheroperanden
.DATA
Multiplikand db 50
Multiplikator db 12
.CODE
MOV AL,[Multiplikand]   ; AL = 50
MUL [Multiplikator]   ; Byte-Multiplikation mit 12
                      ; -> AX=600, CF=1, OF=1 da Ergebnis>255
```

IMUL

Integer Multiplication
vorzeichenrichtige Multiplikation

Beschreibung `IMUL` existiert (ab dem 386) in drei Varianten: Mit einem, zwei oder drei Operanden. Die erste Variante stellt für das Multiplikationsergebnis doppelt so viele Bit zur Verfügung wie die Operanden haben, die beiden letzten nur gleich viele! In den beiden letzten Varianten sind also ernste Fehler möglich! In allen Fällen werden die Flags `CF` und `OF` dann gesetzt, wenn das Ergebnis die Bitbreite der Quelloperanden überschreitet. Für die Varianten mit zwei oder drei Operanden bedeutet dies einen echten Fehler.

Syntax `IMUL` *Multiplikator*
Multiplikator: reg8/16/32/mem8/16/32

Flags | O | | C |

Beschreibung `IMUL` mit einem Operanden führt eine Multiplikation vorzeichenbehafteter Zahlen durch und arbeitet ansonsten wie `MUL`.

Syntax `IMUL` *Operand1*, *Operand2*
Operand1: reg16/32
Operand2: reg16/32/mem16/32|Direktoperand

Flags | O | | C |

Beschreibung `IMUL` mit zwei Operanden führt eine vorzeichenrichtige Multiplikation der beiden Operanden durch und legt das Ergebnis im ersten Operanden ab.

Syntax `CMP` *Operand1*, *Operand2*, *Operand3*
Operand1: reg16/32
Operand2: reg16/32/mem16/32
Operand3: Direktoperand

Beschreibung `IMUL` mit drei Operanden führt eine vorzeichenrichtige Multiplikation des zweiten und dritten Operanden durch und legt das Ergebnis im ersten Operanden ab.

Beispiele

```
; Multiplikation zweier 8 Bit-Speicheroperanden
.DATA
zahl1 dw 1200
zahl2 dd 3000 .CODE
IMUL ECX   ; multipliziert EAX mit ECX, Ergebnis in EDX-EAX IMUL DI, zahl1   ; DI =
```

DIV|IDIV

Unsigned divide|signed Integer div.
Dividieren|vorzeichenrichtiges Div.

Syntax `DIV|IDIV` *Divisor*

Divisor: reg8/16/32/mem8/16/32

Flags —

Beschreibung `DIV` führt eine Division vorzeichenloser und `IDIV` eine Division vorzeichenbehafteter Zahlen durch. Im Befehl wird nur der Divisor explizit als Operand aufgeführt, der Dividend ist immer `AX` bzw. `DX:AX` Je nach Bitbreite des Divisors wird eine Byte– oder eine Wort–Division durchgeführt. Dabei wird jeweils der ganzzahlige Quotient und der Rest separat abgelegt.

Bei der Byte–Division ist der Dividend `AX`. Der ganzzahlige Teil des Divisionsergebnis wird in `AL` und der Rest in `AH` abgelegt.

Bei der Wort–Division ist der Dividend `DX:AX`. Der ganzzahlige Teil des Divisionsergebnis wird in `AX` und der Rest in `DX` abgelegt.

Wenn das|die Zielregister nicht ausreicht um das Ergebnis aufzunehmen (bei kleinen Divisoren) tritt der sog. Divisionsfehler ein. In diesem Fall wird → `INT 0` ausgelöst. Ein Spezialfall des Divisionsfehler ist die Division durch Null. Siehe auch → `MUL, IMUL`

Beispiele

```
; Bsp.1: Division eines Doppelwortes
.DATA
Dividend dd 010017h  ; 65559d
Divisor dw 10h       ; 16d
.CODE
MOV AX,[word ptr Dividend]   ; niederwertiger Teil in AX
MOV DX,[word ptr Dividend+2]   ; höherwertiger Teil in DX
DIV [word ptr Divisor]   ; Division durch 10h
; -> ganzzahliger Teil des Quotienten: AX=1001h (4097d)
; -> Divisionsrest: DX=7
;---
; Bsp.2: Division durch eine negative Zahl
mov ax, 50
mov bl, -3
idiv bl
; -> Divisonsergebnis in AL: AL=F0h (-16 im Zweierkomplement)
; -> Divisionsrest in AH: AH=2
```

16.7 Stackbefehle

PUSH	Push schieben

Syntax `PUSH` *Quelle*
Quelle: reg16/32|mem16/32

Flags —

Beschreibung Kopiert den Quelloperanden an die Spitze des Stack. Dazu wird zunächst `SP` um 2 bzw. 4 dekrementiert und dann der Kopiervorgang nach `SS:SP` ausgeführt. Weitere Erläuterungen. s. → `POP`.

Beispiele `s. POP`

POP	Pop

Syntax `POP` *Ziel*
Ziel: reg16/32|mem16/32

Flags —

Beschreibung Kopiert den Wert an der Spitze des Stack in den Zieloperanden. Anschließend wird `SP` um 2 bzw. 4 erhöht. `POP` wird benutzt, um Werte vom Stack zurückzuholen, die mit `PUSH` dort abgelegt wurden. Die Hauptanwendung ist das Zwischenspeichern („Retten“) von Registerinhalten. Die Befehle `PUSH` und `POP` treten daher in der Regel paarweise auf, es wird dann `SP` automatisch korrekt verwaltet. Auf Speicheroperanden werden `PUSH` und `POP` seltener angewandt.

Beispiele

```
PUSH AX              ;AX retten
PUSH BX              ;BX retten
;
; arbeiten mit AX und BX
;
POP BX               ;BX restaurieren
POP AX               ;AX restaurieren
;---
PUSH DS              ; Datentransport mit PUSH und POP
POP ES
```

16.8 Programmfluß–Steuerungsbefehle

JMP	Jump Springen

Syntax `JMP` *Ziel*

Ziel: Sprunglabel—reg16—mem16

Flags —

Beschreibung Führt einen unbedingten Sprung zum angegebenen Sprunglabel (Marke) durch.

Beispiele

```
        JMP Label1          ; unbedingter Sprung
        .
        .                   ; wird übersprungen
        .
Label1: MOV AX,0FFh
```

JXXX — Jump conditional / Springen wenn

Syntax `JXXX` *Ziel*

Ziel: label(short)

Label im Bereich -128 bis +127 Byte **Flags**—

Beschreibung Führt einen bedingten Sprung zum angegebenen Sprunglabel (Marke) durch, die Sprungweite ist abhängig von den Segmentattributen und kann z.B. auf -128 bis +127 Byte begrenzt (Short–Jump) sein.

Für die Bedingung `XXX` gibt es zahlreiche Möglichkeiten, die in der nachfolgenden Tabelle aufgeführt sind. Bsp.: `XXX=NGE` ergibt den Befehl `JNGE` d.h. "Jump if not greater or equal". Viele bedingte Sprungbefehle sind unter mehreren Mnemonics verfügbar, z.B. ist `JNGE` identisch mit `JL`, "Jump if less". Durch entsprechende Interpretation der Flags wird zwischen Arithmetik mit und ohne Vorzeichen unterschieden. Die arithmetischen bedingten Sprungbefehle werden typischerweise nach einem `CMP` angeordnet. In diesem Fall ist das Sprungverhalten genau so, wie die Namen vermuten lassen.

Beispiele

```
        CMP AX,BX           ; Vgl AX und BX
        JG Label1           ; Sprung wenn AX>BX
        .
        .                   ; wird übersprungen
        .
Label1: MOV AX,0FFh
```

Bedingte Sprünge			
Befehl	Sprungbedingung	Sprungbed. dt.	Flags
Direkte Abfrage von Flags			
JE|JZ	equal|zero	gleich|Null	ZF=1
JNE|JNZ	not equal|zero	ungleich|ungleich Null	ZF=0
JS	signed	Vorzeichen negativ	SF=1
JNS	not signed	Vorzeichen positiv	SF=0
JP|JPE	parity|parity even	gerade Parität	PF=1
JNP|JPO	no parity|parity odd	ungerade Parität	PF=0
JO	overflow	Überlauf	OF=1
JNO	no overflow	kein Überlauf	OF=0
JC	carry	Übertrag	(CF=1)
JNC	no carry	kein Übertrag	(CF=0)
Arithmetik mit Vorzeichen			
JL|JNGE	less|not greater or equal	kleiner	CF ≠ OF
JLE|JNG	less or equal|not greater	kleiner oder gleich	(SF ≠ OF) oder (ZF=1)
JNL|JGE	not less|greater or equal	nicht kleiner	(SF=OF)
JG|JNLE	greater|not less or equal	größer	(SF=OF) und (ZF=0)
Vorzeichenlose Arithmetik			
JA|JNBE	above|not below or equal	oberhalb	(CF=0) und (ZF=0)
JB|JNAE	below|not above or eq.	unterhalb	(CF=1)
JNA|JBE	not above|below or equal	nicht oberhalb	(CF=1) oder (ZF=1)
JNB|JAE	not below|above or equal	nicht oberhalb	(CF=0)

JCXZ

Jump if CX Zero
Springen wenn CX=0

Syntax `JCXZ` *Ziel*

Ziel: label (short)

Flags —

Beschreibung Führt einen Sprung zum angegebenen Sprunglabel (Short–Jump) durch, wenn `CX=0` ist. Dieser Befehl ist in Verbindung mit den `LOOPXX`–Befehlen nützlich. Er kann benutzt werden, um zu erreichen, dass eine `LOOPXX`–Schleife übersprungen wird, wenn `CX=0`.

Beispiele `Siehe LOOPNE`

CALL

CALL
Rufen

Syntax `CALL` *Ziel*

Ziel: Unterprogrammadresse

Flags —

Beschreibung Legt zunächst die Adresse des nächstfolgenden Befehls auf den Stack (→ PUSH) und springt dann zu der angebenen Startadresse des gerufenen Unterprogramms. Das Programm wird also dort fortgesetzt. Wenn im Unterprogramm ein → RET ausgeführt wird, wird die Rücksprungadresse vom Stack geholt und das Programm mit dem nächsten Befehl nach CALL fortgesetzt. Siehe auch → RET

Beispiele

```
; s. RET
```

RET|RETN|RETF

Return|Return NEAR|Return FAR
Zurückkehren|nah|fern

Syntax RET *[Zahl]*

[Zahl]: vorzeichenloser Direktoperand

Flags —

Beschreibung Holt zunächst eine Adresse vom Stack (→ POP) und setzt die Programmausführung dort fort. Verwendung: Rückkehr aus einem Unterprogramm.

RETN holt ein Wort vom Stack und führt einen Near-Sprung aus, RETF holt zwei Worte vom Stack und führt einen Far-Sprung aus. Bei RET entscheidet der Assembler über die Art des Rücksprungs.

Optional kann jede der drei Befehlsformen mit einer vorzeichenlosen Zahl als Argument versehen werden. Diese Zahl wird zu SP addiert, nachdem die Rücksprungadresse vom Stack geholt wurde. Siehe auch → CALL

Beispiele

```
Kopfzeile db 'Programm XY Version 0.9',13,10,'$'
.CODE
CALL Printkopfzeile  ;Unterprogrammaufruf
MOV AH,22
;
Proc Printkopfzeile
MOV ah,9             ;DOS Funktion ''print string''
MOV dx,OFFSET Kopfzeile   ; Adresse von ''Kopfzeile''
INT 21h              ; Zeile ausgeben
RET                  ;Rücksprung, Fortsetz. bei ''MOV AH,22''
ENDP                 ;Ende des Codes der Prozedur
```

INT

Interrupt
Systemaufruf

Syntax INT *[N]*

[N]: Nummer des Aufgerufenen Interrupthandlers

Flags —

Beschreibung Ruft die zugehörige BIOS–Funktion auf und stellt so die Schnittstelle zum Betriebssystem dar. Erlaubte Nummern sind 0 . . . 255. Die weitere Spezifikation der gewünschten Funktion wird durch Übergabe von Parametern in den Registern `AH, AL` etc. vorgenommen. Die größte Gruppe ist `INT 21h` (DOS–Funktionsaufruf).

Beispiele

```
Kopfzeile  DB 'Hallo Assemblerwelt!',13,10,'$'
           MOV AH,9              ;DOS Funktion ''print string''
           MOV DX,OFFSET Kopfzeile   ; Adresse von ''Kopfzeile''
           INT 21h               ; Zeile ausgeben
```

16.9 Stringbefehle

Alle Stringbefehle (→ `MOVS, LODS, STOS, CMPS, SCAS`) haben folgende Gemeinsamkeiten:

- Die Adressierung erfolgt immer über die Registerpaare `DS:SI` und|oder `ES:DI`.
- Die beteiligten Indexregister `DI` und|oder `SI` werden nach der Ausführung des Befehls automatisch verändert. Diese Veränderung hängt von der Bitbreite des Befehls und der Stellung von `DF` ab:

	DF=0	DF=1
Byteoperation	+1	−1
Wortoperation	+2	−2
Doppelwortoperation	+4	−4

- Für die Stringbefehle gibt es eine Befehlsform, aus der nicht ersichtlich ist, ob ein Byte– oder Wortbefehl ausgeführt werden soll, z.B. `MOVS` statt `MOVSB`, `MOVSW` oder `MOVSD` . In dieser Form müssen Operanden angegeben werden, die aber nicht adressiert werden, sondern nur die Bitbreite anzeigen.
- Die Stringbefehle können mit den Wiederholungspräfixen → `REP, REPE, REPNE` versehen werden, so daß Schleifen in einer Befehlszeile programmiert werden können.

MOVSB|MOVSW|MOVSD

Move String Byte|Word|Doubleword
Stringtransport Byte|Wort|Doppelwort

Syntax `MOVSB|MOVSW|MOVSD`

Flags —

Beschreibung Kopiert ein Byte|Wort|Doppelwort von `DS:SI` nach `ES:DI`.

Beispiele

```
        .DATA
titel   DB 'Beispiele zum Assemblerskript',0dh,0ah,'$'
string  DB 50 DUP(?)
bvar1   DB 0
bvar2   DB 0FFh
        .CODE
        ;
        MOV AX,DS            ; Quelle und Ziel liegen
        MOV ES,AX            ; im gleichen Segment
        MOV SI,offset titel   ; Source index = Quellzeiger
        MOV DI,offset string  ; Destination index = Zielzeiger
        MOV CL,32            ; Anzahl
        CLD                  ; DF = 0, -> Inkrement
  l1:   MOVSB                ; move string byte
        DEC CL               ; Dekrement Zaehler
        JNZ L1               ; Schleife wenn cl != 0
        MOV AH,9             ; DOS print string function
        MOV DX,OFFSET string  ; Adresse von ''string''
        INT 21h              ; Ueberschrift ausgeben
        ;---
        MOVS bvar1,bvar2     ; irrefuehrend, BVAR1 und BVAR2
                             ; sind nicht Ziel und Quelle!!
```

LODSB|LODSW|LODSD

Load String Byte|Word|Doubleword
Laden String Byte|Wort|Doppelwort

Syntax `LODSB|LODSW|LODSD`

Flags —

Beschreibung `LODSB|LODSW|LODSD` Kopiert ein Byte|Wort|Doppelwort von `DS:SI` nach `AL|AX|EAX`.

Beispiele

```
        .DATA
titel   DB 'Beispiele zum Assemblerskript',0dh,0ah,'$' .CODE
        ;
        ;
        MOV SI,offset titel  ; Source index = Quellzeiger
        ; ''titel'' liegt im Datensegment,
        ; DS:SI zeigt jetzt auf ''titel''
        CLD                  ; DF = 0, -> Inkrement
        LODSB                ; load string byte
                             ; -> AL='B'
```

STOSB|STOSW|STOSD

Store String Byte|Word|Doubleword
Speichern String Byte|Wort|Doppelwort

Syntax `STOSB|STOSW|STOSD`

Flags —

Beschreibung Speichert das Byte|Wort|Doppelwort in `AL`|`AX` nach `ES:DI`.

Beispiele

```
.DATA
Spruch  DB 'Dreizeitbeschäftigung',0dh,0ah,'$'
.CODE
;
;
MOV AX,DS             ; ES:DI muß auf Ziel zeigen
MOV ES,AX
MOV DI,offset Spruch ; Source index = Quellzeiger
CLD                   ; DF = 0, -> Inkrement
MOV AL,'F'
STOSB                 ; store string byte
                      ; -> 'Freizeit'
```

CMPSB|CMPSW|CMPSD

Compare String Byte|Word|Doubleword
Vergleichen String Byte|Wort|Doppelwort

Syntax `CMPSB`|`CMPSW`|`CMPSD`

Flags O S Z A P C

Beschreibung Vergleicht das Byte|Wort|Doppelwort an `DS:SI` mit dem an `ES:DI` und setzt die Flags wie bei → `CMP`.

Beispiele `s.REPE`|`REPNE`

SCASB|SCASW|SCASD

Scan String Byte|Word|Doubleword
Suchen String Byte|Wort|Doppelwort

Syntax `SCASB`|`SCASW`|`SCASD`

Flags O S Z A P C

Beschreibung Vergleicht das Byte|Wort|Doppelwort in `AL`|`AX`|`EAX` mit dem an `ES:DI` und setzt die Flags wie bei → `CMP`.

Beispiele `s.REPE`|`REPNE`

REP

Repeat
Wiederholen

Syntax `REP` *Stringbefehl*
Stringbefehl: MOVS|STOS

Flags —

Beschreibung REP ist kein eigenständiger Befehl sondern ein Wiederholungspräfix und muß vor einem Stringbefehl stehen. Sinnvoll sind nur MOVS und STOS. REP bewirkt, daß nach jeder Ausführung des nachstehenden Stringbefehls CX dekrementiert und, falls CX≠0, der Stringbefehl erneut ausgeführt wird. Auf diese Art läßt sich in einer Zeile eine bis zu 0FFFFh mal ausgeführte Zählschleife programmieren. → REPNE, SCAS, CMPS.

Beispiele

```
; direktes Schreiben in den Bildschirmspeicher
MOV AX,0B800h        ; Segmentadresse des
MOV ES,AX            ; Bildschirmspeichers nach ES
MOV DI,0             ; DI=0 -> linke obere Ecke
MOV AL,'O'           ; Zeichen
MOV AH,49h           ; Attribut
MOV CX,80            ; Anzahl 80
CLD                  ; DF=0, steigende Adressen
REP STOSW            ; repeat store string word (Schleife)
```

REPE|REPNE

Repeat while equal|not equal
Wiederholen solange gleich|ungleich

Syntax REPE|REPNE *Stringbefehl*

Stringbefehl: SCAS|CMPS

Flags —

Beschreibung REPE|REPNE(identisch sind REPZ|REPNZ) sind keine eigenständige Befehle sondern Wiederholungspräfixe und müssen vor einem der Stringbefehle SCAS oder CMPS stehen. REPE|REPNE bewirkt, daß nach jeder Ausführung des nachstehenden Stringbefehls CX dekrementiert und, falls CX≠0, der Stringbefehl erneut ausgeführt wird. Man hat also eine Zählschleife, wie bei REP. Hier existiert aber ein zweites Abbruchkriterium, nämlich die Gleichheit/Ungleichheit der Operanden:

- REPE–Schleifen werden bei Ungleichheit der Operanden abgebrochen, d.h. wenn ZF=0.
- REPNE–Schleifen werden bei Gleichheit der Operanden abgebrochen, d.h. wenn ZF=1.

→ REPE, SCAS, CMPS.

Beispiele

```
; Aufsuchen eines Zeichens 'A' mit Attribut 07h
; im Bildschirmspeicher
MOV AX,0B800h        ; Segmentadresse des
MOV ES,AX            ; Bildschirmspeichers nach ES
MOV DI,0             ; DI=0 -> linke obere Ecke
MOV AL,'A'           ; Zeichen
MOV AH,07h           ; Attribut
```

```
MOV CX,2000          ; 25*80 = 2000 Worte
CLD                  ; DF=0, steigende Adressen
REPNE SCASW          ; repeat scan string word (Schleife)
```

16.10 Ein- und Ausgabebefehle (Input/Output)

IN	Input from Port Eingabe von Port

Syntax `IN` *Ziel*, *I/O–Adresse*
Ziel: `AL`|`AX`|`EAX`
I/O–Adresse: *Direktoperand*|`DX`

Flags —

Beschreibung Dieser Befehl dient zur Eingabe von Daten über I/O-Ports. Eine unmittelbare Adressierung des Ports ist möglich, wenn die Adresse kleiner als 100H ist. Siehe auch → `OUT`.

Beispiele

```
IN al,20h            ; ISR des 8259A auslesen
;---
MOV DX,3F8h
IN AL,DX             ; Empfängerpufferregister von COM1 lesen
```

OUT	Output from Port Ausgabe von Port

Syntax `OUT` *I/O–Adresse*, *Quelle*
I/O–Adresse: *Direktoperand*|`DX`
Quelle: `AL`|`AX`|`EAX`

Flags —

Beschreibung Dieser Befehl dient zur Ausgabe von Daten über I/O-Ports. Eine unmittelbare Adressierung des Ports ist möglich, wenn die Adresse kleiner als 100H ist. Siehe auch → `IN`.

Beispiele

```
OUT 20h,66h          ; OCW über Port 20h
                     ; an Interruptcontroller senden
;---
MOV DX,3F9h          ; Interrupt Enable Register
IN AL,DX             ; von COM1 lesen
OR AL,03h            ; Bits zur Interruptaktivierung setzen
OUT DX,AL            ; zurückschreiben ins Register
```

16.11 Schleifenbefehle

LOOP	Loop Schleife

Syntax `LOOP` *label (short)*
label (short): Label im Bereich -128 bis +127 Byte

Flags —

Beschreibung Befehl dekrementiert `ECX/CX` und springt zur angegebenen Marke, falls `ECX/CX`≠`0`. Der Befehl dient der einfachen Konstruktion von Schleifen und ersetzt die Befehlsfolge `DEC ECX ; JNZ Sprungmarke` → `LOOPE, LOOPNE, LOOPZ, LOOPNZ, JCXZ`

Beispiele

```
        ; Warteschleife mit NOP--Befehlen (=No Operation)
        Procedure Warten
        ; Parameter: ECX = Anzahl der Warteschleifen
        ; Rückgabe: keine
        JCXZ L2              ; Prozedur verlassen wenn CX=0
L1:     NOP
        LOOP L1              ; Warteschleife
L2:     return
```

LOOPE (LOOPZ)	Loope while equal (Zero) Schleife solange gleich (Null)

Syntax `LOOPE|LOOPZ` *label (short)*
label (short): Label im Bereich -128 bis +127 Byte

Flags —

Beschreibung Befehl dekrementiert `ECX` und springt zur angegebenen Marke, falls

1. `ECX`≠`0`
2. `ZF=1`

Der Befehl dient der einfachen Konstruktion von Schleifen mit zwei Abbruchkriterien. Siehe auch → `LOOP, LOOPNE, LOOPNZ`

Beispiele `s. LOOPNE`

LOOPNE (LOOPNZ)	Loope while not equal (not Zero) Schleife solange ungleich (nicht Null)

Syntax `LOOPNE|LOOPNZ` *label (short)*
label (short): Label im Bereich -128 bis +127 Byte

Flags —

Beschreibung Befehl dekrementiert `ECX` und springt zur angegebenen Marke, falls

1. `ECX`≠`0`
2. `ZF=0`

Der Befehl dient der einfachen Konstruktion von Schleifen mit zwei Abbruchkriterien. Siehe auch → `LOOP, LOOPNE, LOOPNZ`

Beispiele

```
        ; Suche nach einem Punkt in einem Dateinamen
        ; Abbruch wenn Punkt gefunden oder 8 Zeichen untersucht
        ; DI zeigt auf Dateinamen
        MOV CX,8
        L1: CMP [DI],'.'     ; Zeichen = '.' ?
        INC DI               ; Zeiger weiterruecken
        LOOPNE L1            ; Schleife
   l2:  return
```

16.12 Prozessorkontrollbefehle

CLD|STD	Clear|Set Directionflag Löschen|Setzen Richtungsflag

Syntax `CLD|STD`

Flags D

Beschreibung Diese Befehle dienen zur direkten Manipulation des Directionflags (Richtungsflag), und werden benutzt um Stringoperationen vorzubereiten.

- CLD : löscht das Directionflag, `DF=0` → Autoinkrement bei Stringoperationen
- STD : setzt das Directionflag, `DF=1` → Autodekrement bei Stringoperationen

Beispiele `s. REP`

CLI|STI	Clear|Set Interruptflag Löschen|Setzen Unterbrechungsflag

Syntax `CLI|STI`

Flags I

Beschreibung Diese Befehle dienen zur direkten Manipulation des Interruptflags (Unterbrechungsflag). Bei gesetztem Interruptflag sind externe Interrupts zugelassen, sonst nur NMI.

- CLI : löscht das Interruptflag, `IF=0`
- STI : setzt das Interruptflag, `IF=1`

Beispiele

```
; Interruptvektor Timerinterrupt neu setzen
CLI                  ;Interrupts während Manipulation an
                     ;Interrupt Vektoren Tabelle sperren
                     ; Besser: DOS-Funktion 25h benutzen!
MOV [32],DX
MOV AX,CS
MOV [34],AX
STI                  ; Interrupts wieder zulassen
```

CLC|STC|CMC

Clear|Set|Complement Carry
Löschen|Setzen|Komplementieren des Carryflags

Syntax `CLC|STC|CMC`

Flags C

Beschreibung Diese Befehle dienen zur direkten Manipulation des Carryflags.

- CLC : löscht das Carryflag, `CF=0`
- STC : setzt das Carryflag, `CF=1`
- CMC : komplementiert das Carryflag, `CF=1-CF`

Beispiele

```
fehler:  STC   ;Fehlerfall, Rückgabe der Information im Carry
         RET
    ok:  CLC   ;kein Fehler
         RET
```

Literaturverzeichnis

Aktuelle Titel

[1] R. Backer, **Programmiersprache Assembler**, rororo 1993
Eine strukturierte Einführung.

[2] Podschun **Das Assemblerbuch**, Addison-Wesley 1999
Umfangreiches und anspruchsvolles Werk, enthält ein eigenes Assembler–Entwicklungssystem.

[3] Rohde, J.: **Assembler ge-packt**, mitp-Verlag, Bonn, 2001
Kompaktes Nachschlagewerk, enthält auch MMX-, SSE- und SSE2-Befehle.

[4] E.-W. Dieterich, **Turbo Assembler**, Oldenbourg 1999
Systematische Einführung einschließlich der Schnittstellen zu Pascal und C.

[5] W.Link **Assembler-Programmierung**, Franzis 2000.

[6] Müller, Oliver: **Assembler-Referenz** FRANZIS-Verlag 2000.

[7] Erdweg, J: **Assembler Programmierung mit dem PC**, Vieweg 1992,
Kurzgefasstes aber systematisches Einführungswerk.

Ältere Titel

[8] T.Swan, **Mastering Turbo Assembler**, Hayden Books, 1989
Ausgezeichnetes, gut lesbares und umfangreiches englischsprachiges Werk.

[9] P. Heiß, **PC-Assemblerkurs**, Heise 1994
Einführung in Kursform, an Themen orientiert, u.a. VGA-Programmierung, BIOS, DOS, EMS, XMS u.a.m.

[10] T.Swan, **Mastering Turbo Assembler**, Hayden Books, 1989
Ausgezeichnetes, gut lesbares und umfangreiches englischsprachiges Werk.

[11] D. Bradley, **Programmieren in Assembler**, Hanser 1984/86
Veraltet (nur 8086/88) aber gut verständlich, Autor ist absoluter Insider.

[12] Letzel, Meyer, **TASM – Der Turbo Assembler von Borland**, Thomson Publ. 1994,
Unkonventionell im Aufbau, Befehle kurz, viele Beispiele, TSR ausführlich.

[13] Hummel, **Assemblerprogrammierung**, tewi 1993
Vermittelt den Stoff ausschließlich an 9 umfangreichen Beispielen („learning by doing“), Schwerpunkt: Betriebssystem.

[14] P. Norton, J. Socha **Peter Nortons Assemblerbuch**, Markt und Technik
Wenig systematisch aber leicht verständlich und detailliert.

bibitembackerprof R. Backer **Professionelle Assemblerprogrammierung**, Franzis 1994
Das Buch ist nach Kontexten gegliedert: Grafik, Maus, TSR, Protected Mode, EMS, Windows.

[15] Wohak, Maurus **80x86/Pentium Assembler**, Thomson Publ. 1995
Aktuelles und umfassendes Werk, Themen sind u.a. Unterschiede der Prozessoren, Coprozessor, Windows, MASM.

[16] Borland, **Turbo Assembler**, Referenz– und Benutzerhandbuch, Borland GmbH.

[17] Hogan, Thom **Die PC–Referenz für Programmierer**, Systhema Verlag 1992
Nachschlagewerk mit den wichtigsten Informationen über BIOS, DOS, Hardware usw. in Tabellenform. Sinnvolle Ergänzung zu allen Titeln über Assemblersprache.

[18] Hogan, Thom **The Programmer's PC–Sourcebook**, Microsoft Press 1991
Englischsprachiges Original des o.a. Titels.

Index