

FUNDAMENTALS OF DIGITAL COMPUTERS

Kurs: Tim00agr/ BA Mannheim
Dozent: Rachid Elyoussfi
SS 02

Auszug von Kapitel 3 – 6 aus:

Programmiersprache Assembler

—

Eine strukturierte Einführung

Reiner Backer, ISBN: 3-499-19249-7
rororo Verlag

3 MIKROPROZESSOR-TECHNIK

Dieses Kapitel soll der Architektur eines Computers gewidmet sein und behandelt den grundlegenden Aufbau eines Mikroprozessors an Hand des 8086-Prozessors.

Vielleicht werden Sie jetzt denken, dass der 8086-Prozessor veraltet ist und im PC-Bereich nur noch ein Schattendasein fristet. Sollten Sie dieser Meinung sein, kann ich Ihnen nur zustimmen. Die Nachfolgeprozessoren, wie der 80286-, 80386-Prozessor und höher, sind durch die verbesserte Architektur und die höhere Taktfrequenz leistungsfähiger als der Vorläufer 8086. Ab dem 80286 werden auch Multitasking-Betriebssysteme wie OS/2 unterstützt. Da wir aber unter dem Betriebssystem MS-DOS arbeiten - auch die Benutzeroberfläche Windows setzt auf MS-DOS auf - werden die erweiterten Eigenschaften höherentwickelter Prozessoren nicht ausgenutzt. Auch der Hochleistungsprozessor 80486 läuft unter MS-DOS nur als schneller 8086. Die meisten Hochsprachen-Compiler, ob nun für C, Cobol oder Pascal, erzeugen beim Compilieren einen Maschinencode nur für den 8086-Prozessor. Die erweiterten Befehle der Nachfolgeprozessoren werden somit nicht genutzt. Sie brauchen also keine Angst zu haben, etwas Veraltetes lernen zu müssen.

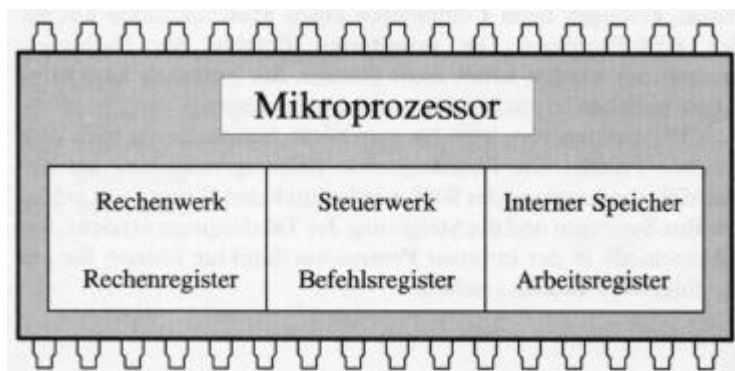
Alle Prozessoren vom 8086 bis zum 80486 funktionieren nach dem gleichen Prinzip. Die fortschreitende Leistungssteigerung bei den Nachfolgeprozessoren des 8086 wurde durch den Einsatz von größeren Bus-Systemen und die Steigerung der Taktfrequenz erreicht. Die Unterschiede in der internen Prozessorarchitektur können Sie aus der folgenden Tabelle ersehen:

Intel Prozessor		8086	80286	80386-DX	80486-DX
Registergröße	(Bit)	16	16	32	32
Adress-Bus	(Bit)	20	24	32	32
Daten-Bus	(Bit)	16	16	32	32
Adressierbarer Speicher	(MB)	1	16	4096	4096
Taktfrequenz	(MHz)	4 - 8	8 - 16	20 - 40	33 - 66
Coprozessor	(integriert)	nein	nein	nein	ja

044a.jpg

3.1 Architektur eines Computers

Seit Beginn des Computerzeitalters in den fünfziger Jahren hat sich bis zum heutigen Tag die Struktur der Rechner im Prinzip nicht verändert. Ein Computer arbeitet nach einem vorgegebenen Programm Anweisungen ab. Um diese Anweisungen durch die Maschine umsetzen zu können, benötigt er als „Manager“ den Mikroprozessor. Diese Art „Herz“ des Computers muss die reibungslose Abarbeitung des laufenden Anwendungsprogramms und die Steuerung des gesamten Systems gewährleisten. Den Mikroprozessor, ein kleines Siliziumplättchen, auf dem sich mehr als eine Million elektrischer Schaltkreise befinden, schauen wir uns näher an:



044b.jpg

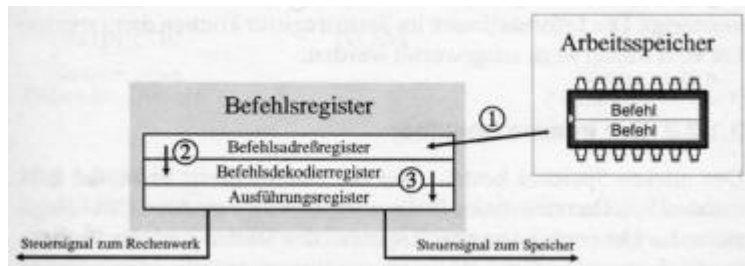
Das Rechenwerk, das Steuerwerk und der interne Speicher bilden den eigentlichen Mikroprozessor. Jeder dieser drei Bauteile des Prozessors erfüllt bei der Abarbeitung eines Programms bestimmte Teilaufgaben. Um diese Aufgaben im Prozessor bearbeiten zu können, müssen die Anweisungen und Befehle im Prozessor gespeichert werden. Daher besitzen das Rechenwerk, das Steuerwerk und der interne Speicher des Prozessors eigene Speicherbausteine, die sogenannten Register. Mit diesen Registern kann der Prozessor für ihn wichtige Informationen auch ohne Hilfe des Arbeitsspeichers behalten. Natürlich könnte man für die Speicherung auch den Arbeitsspeicher benutzen. Die Register haben aber den Vorteil, dass sie sich im Prozessor selbst befinden. Dadurch kann der Mikroprozessor blitzschnell auf die in den Registern zwischengespeicherten Anweisungen und Daten zugreifen.

3.1.2.1 Das Steuerwerk

Das Steuerwerk ist die Kommandozentrale des Prozessors.

Um so richtig kommandieren zu können, braucht auch das Steuerwerk entsprechende Anweisungen. Diese Befehle sagen ihm, was es als nächstes tun soll. Dazu holt sich das Steuerwerk Daten aus dem Arbeitsspeicher. Das Steuerwerk liest den Befehl vom Anwendungsprogramm aus dem Speicher ein, dekodiert diesen Befehl und entscheidet, was zu tun ist.

Um einen Befehl aus dem Speicher lesen zu können, benötigt das Steuerwerk sein Befehlsregister. Dieses Befehlsregister besteht wiederum aus drei Registern. Das erste Register des Befehlsregisters ist das Befehlsadressregister. In ihm befindet sich die Adresse (1) des nächsten auszuführenden Befehls aus dem Arbeitsspeicher.



045.jpg

Der Prozessor bzw. das Steuerwerk holt sich nun aus der angegebenen Adresse im Befehlsadressregister den Befehl aus dem Arbeitsspeicher. Dieser Befehl (2) wird in das nächste Register, das Befehlsdekodierregister, eingestellt. Im Dekodierregister wird der Befehl analysiert, und es wird geprüft, ob weitere Operanden zur Ausführung des Befehls eingelesen werden müssen. Nach der Analyse des Befehls wird dieser an das Ausführungsregister (3) übergeben. Hier entscheidet nun das Steuerwerk, ob zur Ausführung des Befehls weitere Elemente des Prozessors benötigt werden. Bei einer Addition beispielsweise wird das Steuerwerk die eingelesenen Daten an das Rechenwerk übergeben und gleichzeitig das Signal für die Durchführung der Addition erteilen. Bei der Anforderung, Daten aus dem Arbeitsspeicher in den internen Speicher des Prozessors zu transferieren, wird das Steuerwerk die angeforderten Werte in die Allzweckregister des internen Speichers übermitteln.

Das Steuerwerk koordiniert also den gesamten Ablauf des Datenhandlings und verteilt die Aufgaben zur Durchführung eines Befehls an die restlichen Bauteile des Prozessors.

3.1.2.2 Das Rechenwerk

Wie der Name schon sagt, ist das Rechenwerk für die Durchführung der mathematischen Berechnungen zuständig. Hierbei werden jedoch nur die Grundrechenoperationen, Addition, Subtraktion, Multiplikation und Division, unterstützt.

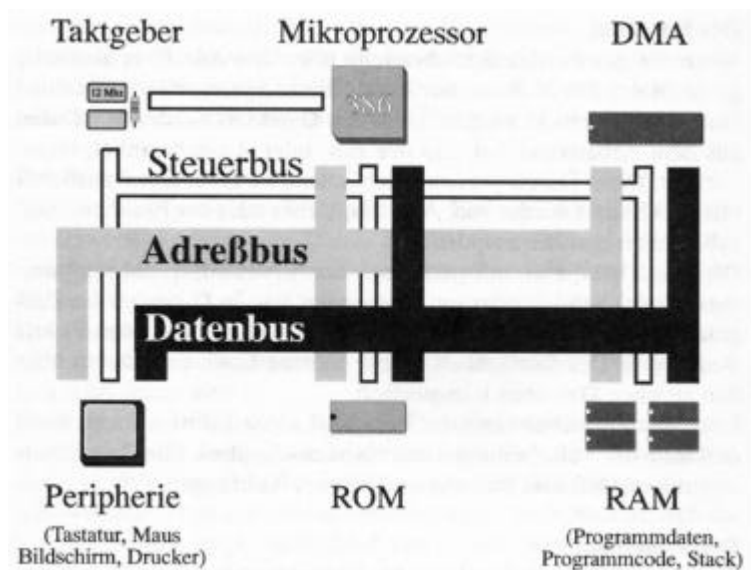
Alle anderen Rechenoperationen werden aus den Grundrechenarten abgeleitet. Weiterhin bearbeitet das Rechenwerk alle Logikprüfungen. Abfragen, wie beispielsweise *Wenn Zaehler größer 100*, werden vom Rechenwerk auf *richtig* oder *nicht richtig* geprüft. Das Ergebnis der Prüfung wird vom Rechenwerk über ein Statusregister angezeigt. Die Informationen im Statusregister können dann wiederum vom Steuerwerk ausgewertet werden.

3.1.2.3 Der interne Speicher

Der interne Speicher besteht, wie der Name bereits vermuten lässt, aus den Speicherstellen des Prozessors, den so genannten CPU-Registern. Im Unterschied zu den Registern des Steuer- und des Rechenwerkes können die CPU-Register vom Programmierer angesprochen werden. Die CPU-Register ermöglichen es dem Programmierer, Daten außerhalb des langsamen Arbeitsspeichers zu speichern und zu verarbeiten. Weiterhin dienen diese Register zur Parameterübergabe an Unterprogramme und an das Betriebssystem selbst.

3.1.2.4 Die Bus-Architektur

Damit der Mikroprozessor Befehle und Daten vom Arbeitsspeicher verarbeiten kann, müssen diese durch ein Verbindungsmittel zum Prozessor selbst übertragen werden. Weiterhin muss eine Verbindung zu den Ein- und Ausgabegeräten vorhanden sein, um mit dem Anwender in Kontakt treten zu können. Diese Verbindungswege im Computer werden als Bus-System bezeichnet. In der folgenden Abbildung sind die „Verbindungsstraßen“ eines Computers dargestellt.



047.jpg

Der Adressbus

Das Steuerwerk im Mikroprozessor bekommt während der Laufzeit eines Programms Informationen vom Anwendungsprogramm nur über Adressen. Sie erinnern sich, eine Adresse spezifiziert eine Speicherstelle im Arbeitsspeicher. Um an den Inhalt dieser Speicherstelle zu gelangen, wird die Adresse durch das Steuerwerk auf den Adressbus gelegt. Der Adressbus zeigt dann auf den betreffenden Speicherinhalt, der zur Verarbeitung ansteht. Dieser Speicherinhalt kann sich dabei im Arbeitsspeicher oder in der Ein-/Ausgabeeinheit befinden. Auch die Tastatur oder der Bildschirm besitzen Speicherbausteine.

Die Anzahl der Leitungen auf dem Adressbus bestimmt dabei die Anzahl der Adressen, die vom Mikroprozessor angesprochen werden können. Der 8086 besitzt 20 Leitungen auf dem Adressbus. Mit diesen 20 Leitungen können nach der Binär-Logik 220 Adressen verwaltet werden, was einer Gesamtzahl von einem Megabyte (MByte) entspricht.

Die Adressierung geht immer vom Steuerwerk aus, der Adressbus arbeitet daher direktional nur in einer Richtung.

Der Datenbus

Wenn die gewünschte Speicherstelle über den Adressbus ausfindig gemacht worden ist, kann der Inhalt dieser Speicherstelle über den Datenbus verschickt werden. Über den Datenbus werden alle Daten aus dem Arbeitsspeicher und der Ein- oder Ausgabeeinheit transportiert. Diese Daten werden anschließend im Prozessor verarbeitet oder bei Bedarf wieder zum Arbeitsspeicher oder der Ein- bzw. Ausgabeeinheit zurücktransportiert. Ob es sich bei diesen Informationen um Anweisungen des Programms (Code) handelt oder um Operanden, die im Datenteil des Programms definiert wurden (Daten), ist bei Mikroprozessoren ohne Bedeutung. Der Einfachheit halber werden Code und Daten über den gleichen Datenbus transportiert.

Über den Datenbus kann der Prozessor somit Informationen lesen und nach der Verarbeitung wieder zurückschreiben. Die Datenübertragung verläuft hier bidirektional in zwei Richtungen.

Der Steuerbus

Der Steuerbus übernimmt bei der Datenübertragung im Computer die Rolle eines Verkehrspolizisten auf einer stark befahrenen Kreuzung. In welcher Richtung die Informationen transportiert werden, wird über den Steuerbus geregelt. Natürlich wird der Steuerbus hierbei von anderen Bauteilen mit entsprechenden Informationen versorgt.

Eine Anweisung, eine Speicherstelle zu lesen oder zu beschreiben, wird vom Steuerwerk des Prozessors über den Steuerbus weitergegeben. Ist der Arbeitsspeicher nun beispielsweise zur Ein- oder Ausgabe von Speicherinhalten nicht bereit, wendet sich auch dieser an den Steuerbus. Der Steuerbus sperrt dann den Zugriff auf den Arbeitsspeicher. Ist der Arbeitsspeicher wieder frei, wird die Blockierung vom Steuerbus aufgehoben, und es können weiter Daten ausgetauscht werden.

Der Steuerbus koordiniert also die ganzen zeitlichen Abläufe, damit es nicht zu einem Stau oder noch schlimmer zu einem Datencrash kommen kann.

3.1.3 Der Aufbau des Intel 8086-Prozessors

Der Intel 8086-Prozessor besteht im inneren Aufbau aus zwei getrennten Funktionseinheiten. Die erste Komponente ist die Bus Interface Unit (BIU), welche für die Verwaltung des Datentransportes zuständig ist. Die zweite Einheit wird durch die Execution Unit (EU) gebildet, die die Maschinenbefehle ausführt.

Bus Interface Unit (BIU)

Die Bus Interface Unit (BIU) leitet die Ein- und Ausgabe der vom Steuerwerk angeforderten Daten. Die BIU tritt dabei als Vermittler zwischen dem Steuerwerk und dem Bussystem auf, daher auch der Name *Bus-Schnittstellen-System*.

Die Hauptaufgabe der BIU ist die Ausführung und Überwachung aller externen Busoperationen zum Arbeitsspeicher und den Ein- und Ausgabegeräten.

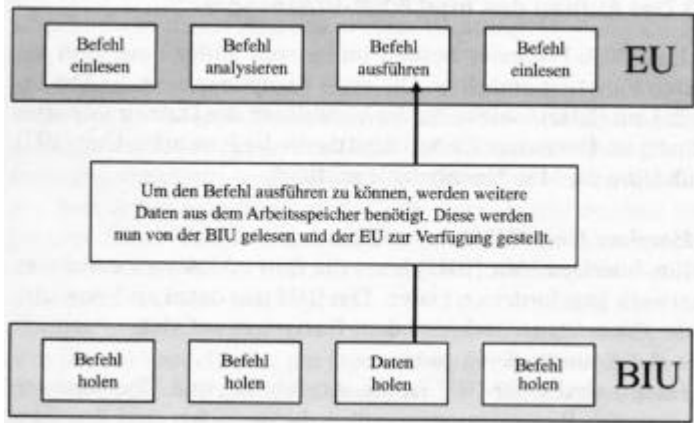
Weiterhin kontrolliert die BIU die Befehlswarteschlange. Werden vom Steuerwerk beispielsweise keine Daten über die BIU angefordert, so füllt die BIU bereits die nächsten Befehle in die Befehlswarteschlange auf. Dieses *Vorlesen* der abzuarbeitenden Befehle wird *als prefetch* (engl.) bezeichnet. Die *vorgelesenen* Befehle werden sequentiell nach aufsteigenden Adressen im Arbeitsspeicher ausgelesen und in die

Befehlswarteschlange eingestellt. Die BIU führt diese Aufgabe immer dann durch, wenn die zweite Komponente des Prozessors, die für den internen Befehlsablauf verantwortlich ist, einen Befehl verarbeitet.

Wird beispielsweise eine Addition durchgeführt, so übergibt das Steuerwerk die Operanden an das Rechenwerk und gleichzeitig den Befehl zum Addieren. Jetzt wäre die BIU eigentlich arbeitslos, da die EU keinen Auftrag für eine Ein-/Ausgabeoperation an sie stellt. Die BIU nützt nun aber die Zeit und liest in der Zwischenzeit neue Befehle in die Befehlswarteschlange ein.

BIU und EU können somit unabhängig voneinander arbeiten, was eine optimale Ausnutzung des Bus-Systems und eine Vergrößerung der Arbeitsgeschwindigkeit gewährleistet. Die zeitintensiven Befehlszugriffe vom Arbeitsspeicher zum Prozessor während der linearen Programmabarbeitung entfallen.

EU und BIU während der Programmabarbeitung

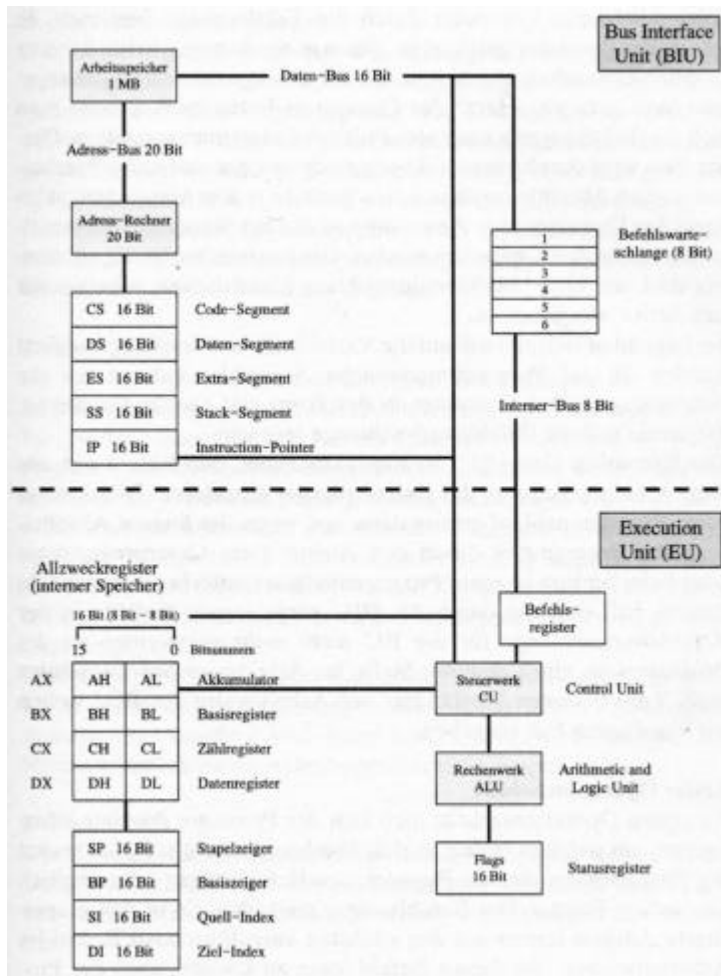


050.jpg

Execution Unit (EU)

Die Execution Unit ist die Verarbeitungseinheit des Prozessors, in der die Maschinenbefehle ausgeführt werden. Die EU holt sich die zu verarbeitenden Maschinenbefehle aus der Befehlswarteschlange. Bevor der Befehl zur Steuereinheit oder Control Unit (CU) kommt, wird dieser Befehl im Befehlsregister analysiert und festgestellt, ob weitere Operanden zur Befehlsausführung eingelesen werden müssen. Sind der Befehl und die dazu benötigten Operanden vollständig eingelesen, wird die Ausführung der Maschineninstruktion im Rechenwerk oder Arithmetic Logic Unit (ALU) ausgeführt.

Die Hauptaufgabe der EU ist also das Einlesen von Befehlen aus der Befehlswarteschlange sowie das Analysieren der Befehle. Benötigt der eingelesene Befehl weitere Operanden, so werden diese über die BIU aus dem Arbeitsspeicher angefordert:



051.jpg

3.1.4 Die Befehlsabarbeitung

Jeder Prozessor arbeitet die ihm übertragenen Befehle in einer festen zeitlichen Reihenfolge nacheinander ab. Die zeitliche Reihenfolge wird beim Computer durch die Taktfrequenz bestimmt. Je höher der Prozessor getaktet ist, desto mehr Arbeitsschritte kann er in einer Zeiteinheit abarbeiten. Da wir den Prozessor im übertragenen Sinn gern als „Herz“ des Computers bezeichnen, könnte man sich die Taktfrequenz auch als „Puls“ der Maschine vorstellen. Dieser Puls wird durch einen Taktgenerator erzeugt und aufrechterhalten und als Maschinenzyklus bezeichnet. In jedem Maschinenzyklus kann der Prozessor eine Anweisung an die zur Steuerung notwendigen Bauteile des Computers senden. Die Einzelschritte, die notwendig sind, um eine Maschineninstruktion abzuarbeiten, schauen wir uns dazu näher an.

Im folgenden Beispiel soll auf die Variable *Zähler* der Wert 4 addiert werden. In der Programmiersprache Assembler müssen wir die Anweisung an den Computer in der Form `add zaehler,4` codieren. Weiterhin soll die Befehlswarteschlange leer sein.

Die Execution Unit (EU) ist nun gezwungen, den Befehl erst aus dem Arbeitsspeicher in das Befehlsregister einzulesen. Dies kommt beim Programmablauf immer dann vor, wenn die lineare Abarbeitung des Programms durch den Aufruf eines Unterprogramms oder beim Sprung zu einer Programmadresse unterbrochen wird. In diesem Fall sind die durch die BIU vorgelesenen Befehle in der Befehlswarteschlange für die EU nicht mehr verwertbar, da das Programm an einer anderen Stelle im Arbeitsspeicher fortgeführt wird. Zum besseren Verständnis der Arbeitsweise der BIU wollen wir von diesem Fall ausgehen.

Erster Operationsschritt

Im ersten Operationsschritt muss sich der Prozessor darüber informieren, um welchen Befehl es sich überhaupt handelt. Dafür besitzt der Prozessor ein eigenes Register, den Befehlszeiger oder englisch Instruction-Pointer. Der Befehlszeiger zeigt über die in ihm gespeicherte Adresse immer auf den nächsten auszuführenden Befehl im Arbeitsspeicher. Um diesen Befehl lesen zu können, muss der Prozessor diesen Befehl in sein Befehlsregister laden.

Dazu stellt der Prozessor die Adresse im Befehlszeiger in den Adressrechner. Dieser Schritt ist notwendig, da der Befehlszeiger leider nur eine Breite von 16 Bit hat. Zur Adressierung des ein Megabyte großen Arbeitsspeichers wird aber eine 20-Bit-Größe benötigt. Deshalb wird zusätzlich zum Befehlszeiger noch das Code-Segment in den Adressrechner geladen.

Das Code-Segment zeigt auf den Anfang des Programmteils im Arbeitsspeicher, in dem sich die Anweisungen des Programms befinden. Aus dem Anfangspunkt des Code-Segments und der entsprechenden Entfernung zum Befehl selbst, die im Befehlszeiger gespeichert ist, wird die 20-Bit-Adresse vom Adressrechner gebildet. Wie dieses funktioniert, werden wir uns gleich näher anschauen.

Die entsprechende Speicheradresse wird anschließend über den Adressbus angesprochen, und gleichzeitig wird das Signal zur Datenübertragung an den Steuerbus gegeben.

Zweiter Schritt

Im zweiten Operationsschritt überprüft der Prozessor die zur Datenübertragung benötigten Bauteile wie Hauptspeicher und Datenbus. Ist der Hauptspeicher oder der Datenbus nicht bereit, wird vom Prozessor ein Warteschritt eingeschoben. Zusätzlich werden alle anderen an der Datenübertragung beteiligten Bauteile über den Steuerbus darüber informiert, dass sich die Datenübernahme aus dem Speicher verzögert. Der Prozessor bleibt dann so lange im Wartezustand, bis er das Signal erhält, dass alle Komponenten für die Transaktion bereit sind. Dieses Warten des Mikroprozessors wird als *Wait-State* (engl.) bezeichnet.

Beim Computerkauf stolpert man öfter über diesen Ausdruck. Wenn Sie in der Produktbeschreibung in den Ausführungen zum Arbeitsspeicher die Angabe *0 Wait-States* finden, zeugt dies von schnellen Speicherbausteinen im Arbeitsspeicher des Computers.

Dritter Schritt

Der Arbeitsspeicher, der Datenbus und der Prozessor stehen für die Datentransaktion bereit. Ob die Daten vom Arbeitsspeicher in den Prozessor oder umgekehrt transportiert werden, regelt der Steuerbus.

Operationszyklus

Die drei aufgeführten Operationsschritte ergeben einen Operationszyklus zur Datenübertragung vom Arbeitsspeicher zum Prozessor oder umgekehrt vom Prozessor zum Arbeitsspeicher.

Für die Ausführung eines Operationszyklus können maximal noch zwei Operationsschritte hinzukommen. Die beiden Operationsschritte werden noch zusätzlich benötigt, wenn das Rechenwerk im Operationszyklus beansprucht werden muss. Dies kommt beispielsweise bei arithmetischen Anweisungen oder bei Vergleichen vor.

Abhängig vom Befehl selbst können so bis zu vier verschiedene Operationszyklen durchlaufen werden.

Mögliche Operationszyklen

Holen:	Der Prozessor holt sich über den Arbeitsspeicher einen Maschinenbefehl vom Programm in das Befehlsregister.
Lesen:	Der Prozessor benötigt einen Speicheroperanden, der im Befehl mit verarbeitet werden muss. In unserem Fall ist dies die Variable <i>Zaehler</i> .
Ausführen:	Der Maschinenbefehl wird im Prozessor durch das Rechenwerk ausgeführt.
Schreiben:	Wurde ein Speicheroperand bei der Ausführung des Befehls verändert, muss der Speicheroperand wieder in den Arbeitsspeicher zurückgeschrieben werden.

In unserem Beispiel (Addiere zur Variable *Zaehler* den Wert 4) werden, wie nachfolgend dargestellt, alle vier Operationszyklen benötigt.

Maschinenbefehl	Zyklen
add zaehler, 4	Holen + Lesen + Ausführen + Schreiben

Jeder Maschinenbefehl besteht aus einem bis fünf Operationszyklen, jeder Operationszyklus besteht aus drei bis fünf Operationsschritten. Um diesen Maschinenbefehl umzusetzen, braucht der Prozessor 17 Operationsschritte. Ein Operationsschritt oder Takt beträgt bei einer Taktfrequenz von 10 Megahertz ca. 0,1 Millisekunden oder 100 Nanosekunden. Die Abarbeitung dieses Befehls durch den Computer dauert also 100 Nanosekunden * 17 Operationsschritte, was einer Arbeitszeit von 1,7 Millisekunden entspricht. Die Nachfolger des 8086-Prozessor arbeiten dabei sehr viel schneller, da über den weiterentwickelten breiteren Systembus mit einem Operationsschritt mehrere Bytes aus dem Arbeitsspeicher über den Datenbus transportiert werden können. Durch die erhöhte Taktfrequenz werden die Daten in einer kürzeren Zeiteinheit transportiert und verarbeitet.

3.2 Übung 3

1. Aufgabe

- Aus welchen Bauteilen besteht ein Prozessor?
- Wie heißen die Verbindungswege im Computer?

2. Aufgabe

Der Intel 8086-Prozessor besteht intern aus zwei getrennten Funktionseinheiten. Welche Funktionseinheiten sind dies, und welche Aufgabe haben sie?

3.3 Lösung 3

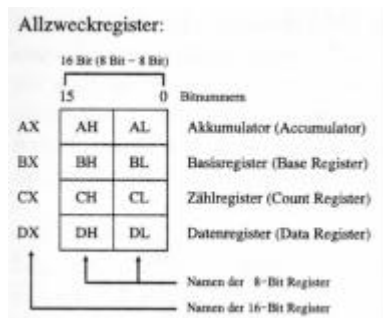
- Der Prozessor besteht aus Rechenwerk, Steuerwerk und einem internen Speicher.
 - Die Verbindungswege im Computer werden als Bussystem bezeichnet. Der Computer verfügt über den Adressbus, den Datenbus und den Steuerbus.
- Der Intel 8086-Prozessor besteht intern aus der BIU und der EU.
BIU (Bus Interface Unit = Busschnittstellensystem), eine Komponente des 8086-Prozessors, die für alle externen Busoperationen zum Arbeitsspeicher und den Ein-/Ausgabegeräten verantwortlich ist.
EU (Execution Unit), die zweite Komponente des 8086-Prozessors, die die Maschinenbefehle ausführt.

4 DIE CPU-REGISTER

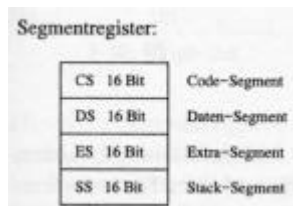
Dieses Kapitel behandelt die Bauteile des Prozessors, die vom Programmierer angesprochen und verwaltet werden. Die CPU-Register, das „Handwerkszeug“ eines Assemblerprogrammierers, sollen auch hier anhand des 8086-Prozessors vertieft werden.

Jeder Prozessor benötigt interne Speicherbausteine, um Daten bei der Abarbeitung eines Programms zwischen den Bauteilen des Prozessors transportieren und Zwischenspeichern zu können. Diese Speicherbausteine werden Register genannt. Die Register befinden sich im Inneren des Prozessors, so dass ein blitzschneller Zugriff auf die Daten gewährleistet ist. Mit diesen Speicherbausteinen oder besser Registern werden die Befehle selbst oder die zur Adressierung des Programms notwendigen Daten festgehalten und zwischengespeichert. Im Unterschied zu anderen Speicherbausteinen brauchen wir die Register nicht mit einer Zahl zu betiteln, da für jedes Register ein Name vergeben wurde.

In der nachfolgenden Abbildung sehen Sie alle Register des 8086-Prozessors, die vom Programmierer angesprochen und verwaltet werden.



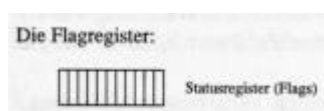
057a.jpg



057b.jpg



057c.jpg



057d.jpg

Mit Hilfe dieser Register und den 90 Grundbefehlen des 8086-Prozessors können Sie vom einfachen Berechnungsprogramm bis hin zur komplizierten Datenbank alles realisieren, was ein Computer leisten kann. Die Register sind die internen Speicherstellen des Prozessors, der Arbeitsspeicher des Prozessors sozusagen, mit dem Vorteil, dass die Daten in diesen Registern ohne zeitintensive Systembus-Zugriffe transportiert werden können.

Da Arbeitsspeicher und Prozessor zwei getrennte Bauteile darstellen, müssen die Daten erst über die Leitungen des Computers transportiert werden. Um diese Zeitverzögerung zu umgehen, werden wir die internen Speicherstellen des Prozessors bzw. die Register so oft wie möglich nutzen. Mit Hilfe dieser Register ist es möglich, Daten außerhalb des langsamen Arbeitsspeichers zu verarbeiten und diese Daten nahe am Prozessor zu halten.

Der Prozessor wird es Ihnen durch eine schnelle Abarbeitung des Programms danken. Wie die Register miteinander arbeiten und welche spezielle Aufgabe jedem Register zugeteilt ist, wollen wir uns nun näher anschauen.

4.1 Allzweckregister

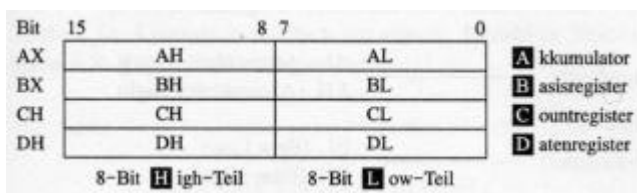
Die Allzweckregister setzen sich aus dem Akkumulator (AX), dem Basisregister (BX), dem Zählregister (CX) und dem Datenregister (DX) zusammen. Mit den Allzweckregistern werden Sie bei der Codierung eines Programms am meisten zu tun haben.

Bei allen Registern des 8086-Prozessors handelt es sich um 16-Bit-Register. Nach der Binärlogik kann ein Wertebereich bis 2¹⁶ in jedem Register gespeichert und verarbeitet werden.

Wenn alle 16 Bit eine Null enthalten, ist dies der niedrigste Wert, den ein Register aufnehmen kann. Der höchste Wert wird dargestellt, indem alle 16 Bit eine Eins aufweisen. Dies entspricht bei vorzeichenlosen Zahlen einem Wertebereich von 0 bis 65.535. Bei Zahlen mit Vorzeichen kann nur ein Wertebereich von -32.768 bis 32.767 dargestellt werden, da das Vorzeichen in der binären Darstellung ein Bit belegt und deshalb zur Darstellung einer Zahl nur noch 15 Bit zur Verfügung stehen.

Möchten Sie einen größeren Wertebereich verarbeiten, so müssen Sie zwei Register zusammenfassen. Mit diesem „selbst gebastelten“ 32-Bit-Register können Sie dann vorzeichenlose Zahlen von 0 bis 4.294.967.295 oder Werte mit Vorzeichen von -2.147.483.648 bis 2.147.483.647 speichern und verarbeiten.

In der Praxis werden am häufigsten 8-Bit-Zugriffe benötigt. Mit 8 Bit lassen sich Werte von -128 bis 127 oder vorzeichenlose Werte von 0 bis 255 darstellen. Das ist zum Rechnen zwar nicht viel. Für die Darstellung des gesamten bearbeitbaren Zeichensatzes ist das jedoch ausreichend, denn alle ASCII-Zeichen nehmen einen Wertebereich von 0 bis 255 ein. Die 8-Bit-Register werden daher besonders bei der Verarbeitung von Zeichenketten verwendet. Dafür sind die Allzweckregister bestens vorbereitet. Alle vier Allzweckregister lassen sich zusätzlich in zwei 8-Bit-Register unterteilen und ansprechen.



059.jpg

Die vier Allzweckregister, mit einer Breite von 16 Bit, sind zusätzlich in einen High- und einen Low-Teil unterteilt. Im Low-Teil werden die niederwertigen 8 Bits angesprochen, also die Bits 0 bis 7. Im High-Teil werden die höherwertigen Bits von 8 bis 15 angesprochen. Dies gilt jedoch nur für die Allzweckregister. Alle anderen Register des Prozessors können nur als 16-Bit-Größen verwaltet werden. Durch die Unterteilung der Allzweckregister in zwei weitere Teile können wir gezielt mit Werten von 8 Bit arbeiten. Dies kommt uns besonders bei der Zeichenverarbeitung zugute.

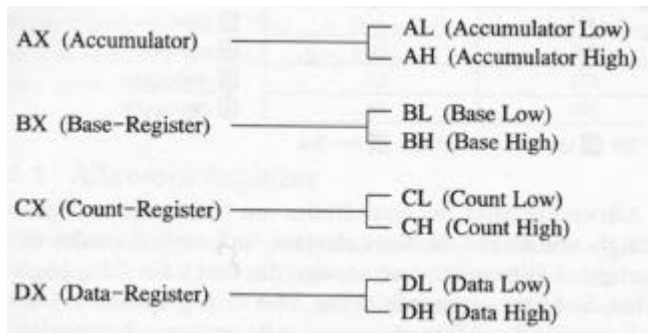
Natürlich kann man ein Zeichen auch mit einem 16-Bit-Register bearbeiten, der High-Teil des Registers würde jedoch nicht verwendet werden. Um dieser Platzverschwendung entgegenzuwirken, sind die vier Allzweckregister nochmals unterteilt. So können Sie die vier 16-Bit-Allzweckregister im Bedarfsfall als acht Register mit 8 Bit ansprechen.

Kennzeichnung

Damit Sie die CPU-Register ansprechen und auf sie zugreifen können, müssen ihnen Namen gegeben werden. Der erste Buchstabe steht dabei als Abkürzung für den englischen Namen des Registers.

Ob Sie das Allzweckregister als Ganzes ansprechen wollen oder nur einen Teilbereich, ist unerheblich. Der Buchstabe A steht für Accumulator, B für das Base-Register, C für das Count-Register und D für Data-Register. Beim zweiten Buchstaben wird jetzt unterschieden, welchen Teil des Allzweckregisters Sie ansprechen wollen. Möchten Sie das Register als Ganzes ansprechen, also als 16-Bit-Register, müssen Sie für den zweiten Buchstaben ein X wählen. Möchten Sie nur den unteren Teil des Registers ansprechen, so tragen Sie für den zweiten Buchstaben ein L (low) und für den höheren Teil des Registers ein H (high) ein.

Die vier 16-Bit-Allzweckregister, die bei Bedarf als acht 8-Bit-Register angesprochen werden können, heißen folgendermaßen:



060.jpg

Die vier Allzweckregister tragen ihren Namen also nicht nach den Anfangsbuchstaben des Alphabets, sondern nach den ihnen zugeordneten Aufgabengebieten. Grundsätzlich können Sie alle Register bei fast allen Befehlen individuell kombinieren. Manche Maschinenbefehle verlangen aber bei der Ausführung eines Befehls ein spezielles Allzweckregister als Operanden. Jedes Allzweckregister hat ein spezielles Aufgabengebiet:

Das AX-Register

Der Akkumulator (AX-Register) wird hauptsächlich bei allen arithmetischen Operationen eingesetzt. Standardmäßig muss es bei der Multiplikation und Division verwendet werden. Dieses Register sollte aber auch bei der Addition und Subtraktion sowie bei logischen Vergleichen eingesetzt werden, da arithmetische Aufgaben dadurch effizienter bearbeitet werden können.

Zu jedem Register möchten wir ein Beispiel bringen.

Für die nachfolgenden Beispiele nehmen wir vieles vorweg. Die ausführliche Dokumentation soll den Befehlsablauf verständlich gestalten. Die bereits jetzt vorgestellten Befehle des Assemblers werden später ausführlich behandelt.

Beispiel

Der Befehl zur Multiplikation ist standardmäßig mit dem Register **AX** verbunden. Um dies besser darlegen zu können, möchten wir eine Variable *Flaeche* mit den beiden Operanden *Laenge* und *Breite* multiplizieren. Der entsprechende Befehl könnte $Flaeche = Laenge * Breite$

lauten. In Assembler müssten wir diesen Befehl in Verbindung mit dem AX-Register wie folgt codieren:

```
Flaeche dw    ?      ; Variable Flaeche mit dw = 2 Bytes definieren
                          ; Die Variable wird mit keinem Wert vorbelegt
Laenge  db    50     ; Variable Laenge mit db = 1 Byte definieren
                          ; Zusätzlich die Variable Laenge mit 50 vorbelegen
Breite  db    60     ; Variable Breite mit db = 1 Byte definieren
                          ; Zusätzlich die Variable Breite mit 60 vorbelegen

mov  al, Laenge      ; Der erste Operand muss sich in AL befinden
                          ; Schiebe dazu den Inhalt der Variablen Laenge in das Register AL
mul  Breite          ; Der zweite Operand muss beim MUL- Befehl selbst mit angegeben werden -
                          ; Das Ergebnis wird dann in AX abgelegt AX = AL * Breite
mov  Flaeche, ax     ; Schiebe den Inhalt des Registers AX, in die Variable Flaeche zurück
```

Das BX-Register

Möchten Sie den Inhalt des Arbeitsspeichers direkt ansprechen, so müssen Sie das BX-Register verwenden. Das Basis-Register mit der Abkürzung BX erlaubt es uns, direkt auf den Arbeitsspeicher zuzugreifen. Das BX-Register wird dazu als eine Art Zeiger auf den Inhalt des Speichers verwendet. Stellen Sie sich das BX-Register dazu als langen Stab vor. Mit Hilfe dieses Stabes kann der Inhalt von Speicherzellen direkt gelesen und beschrieben werden, was dieses Register besonders für die Tabellen- und String-Verarbeitung interessant macht.

Beispiel

Sie möchten eine Zeichenkette nach einem bestimmten Buchstaben durchsuchen. In unserem Fall soll es sich um den Buchstaben *w* handeln. Wird dieser Buchstabe gefunden, soll er durch ein *a* ersetzt werden.

```
String  db    "Wind", 0      ; Variable String mit dem Inhalt Wind definieren
Laenge  equ    $ - String    ; Länge des Strings . "$" steht für die aktuelle Position

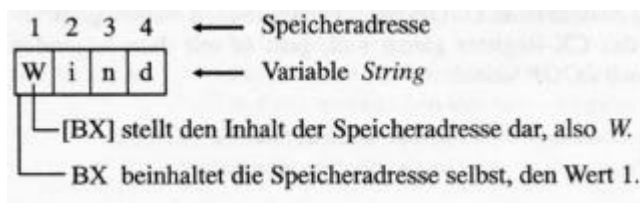
mov  bx, off set String      ; Das BX-Register zeigt auf die Anfangsadresse der Variablen String.
                          ; BX wird also auf die Speicheradresse zeigen, in der das Zeichen W enthalten ist
mov  al, Laenge              ; Länge der Zeichenkette nach AL
```

```

schleife:                ; Wir vereinbaren ein Label
                        ; Vergleiche den Inhalt der Speicherstelle mit dem Buchstaben i
      cmp [bx], "i"
      je gefunden        ; Wenn gleich springe zu dem Label gefunden:
      inc bx             ; Bx um eins erhöhen -> BX = BX + 1
                        ; Eine Speicherstelle weiter - BX zeigt jetzt auf den nächsten Buchstaben
      dec al             ; Eins von der Länge des Strings abziehen -> AL = AL - 1
      cmp al, 0          ; Sind wir schon am Ende des Strings ?
      jne schleife      ; Nein - springe zu dem Label schleife:
      je ende           ; Ja - springe zu dem Label ende:
gefunden:                ; Der Buchstabe wurde gefunden
      mov [bx], "a"     ; Den Buchstaben a zurückschreiben
ende:                    ; Ende Programm

```

Dieses Beispiel hat einiges vorweggenommen. Wie schon erwähnt, möchten wir, dass Sie sich langsam mit der Programmierung in Assembler anfreunden. Die vorgestellten Befehle in diesem Beispiel werden Sie in Assembler-Programmen sehr häufig wiederfinden. Deshalb haben wir dieses Beispiel zum BX-Register etwas erweitert. Im Listing zu diesem Beispiel wurde das BX-Register einmal in geschlossenen eckigen Klammern *[bx]* verwendet und einmal ohne Klammern *bx*. Ohne eckige Klammern wird beim Adressieren eines Speicherbereiches die Speicherstelle selbst angesprochen. Bei der Verwendung von eckigen Klammern sprechen wir den Inhalt dieser Speicherstelle direkt an. Nachfolgende Zeichnung soll dies verdeutlichen.



063.jpg

Das CX-Register

Das Count- oder Zählregister ist der Spezialist, wenn es um den Einsatz von Schleifen geht.

Beispiel

Sie möchten einen Programmschritt fünfmal wiederholen.

In dem Beispiel zum BX-Register haben wir uns unsere Schleife selbst konstruiert, in der Form

```

      mov al, laenge      ; Länge der Zeichenkette nach AL
schleife:                ; Wir vereinbaren ein Label
...
; <Befehle die wiederholt werden>
...
      dec al             ; Eins von der Länge des Strings abziehen -> AL = AL - 1
      cmp al, 0          ; Sind wir schon am Ende des Strings ?
      jne schleife      ; Nein - springe zu dem Label schleife:

```

Das Wiederholen von Befehlssequenzen durch Schleifen kommt in der Programmierung so oft vor, dass die Entwickler der Intel-Prozessoren einen eigenen Befehl dafür erdacht haben, den LOOP-Befehl. Dieser Befehl verwendet standardmäßig das CX-Register und zieht von dessen Inhalt den Wert eins ab. Nach dieser Operation wird das CX-Register mit dem Wert null verglichen. Ist CX ungleich null, wird zu dem beim LOOP-Befehl angegebenen Namen gesprungen. Ist das CX-Register gleich null, geht es mit dem folgenden Befehl nach LOOP weiter:

```

      mov cx, laenge     ; Länge der Zeichenkette nach CX
schleife :              ; Wir vereinbaren ein Label
...
; <Befehle, die wiederholt werden>
...
      loop schleife     ; Solange CX ungleich null springe zu dem Label schleife:
...
; <Weitermachen>
...

```

Das DX-Register

Dieses Register wird besonders in Verbindung mit 32-Bit-Multiplikations- und Divisionsoperationen genutzt. Das DX-Register

wird hier zusammen mit dem AX-Register verwendet.

Beispiel

Bei der Multiplikation wollen wir jetzt 32-Bit-Werte verarbeiten. Wie im ersten Beispiel zum AX-Register verwenden wir wieder die Formel *Flaeche = Laenge * Breite*, mit dem Unterschied, dass wir größere Zahlenwerte benutzen. Die Variable *Laenge* soll mit dem Wert 5000 vorbelegt sein, die Variable *Breite* mit 6000. Das Ergebnis mit dem Wert 30.000.000 würde nicht mehr in ein einzelnes 16-Bit-Register passen.

```

Flaeche dd      ?           ; Variable Flaeche mit dd = 4 Bytes definieren
Laenge  dw      5000        ; Variable Laenge mit dw = 2 Byte definieren
                               ; Variable Laenge mit dem Wert 5000 vorbelegen
Breite  dw      6000        ; Variable Breite mit dw = 2 Byte definieren
                               ; Variable Breite mit dem Wert 6000 vorbelegen

...
...
mov ax, Laenge      ; Der erste Operand muss sich in AX befinden
mul Breite          ; Der zweite Operand muss beim MUL-Befehl angegeben werden - Ergebnis :
                               ; -HIGH-Teil inDX, LOW-Teil inAX.

```

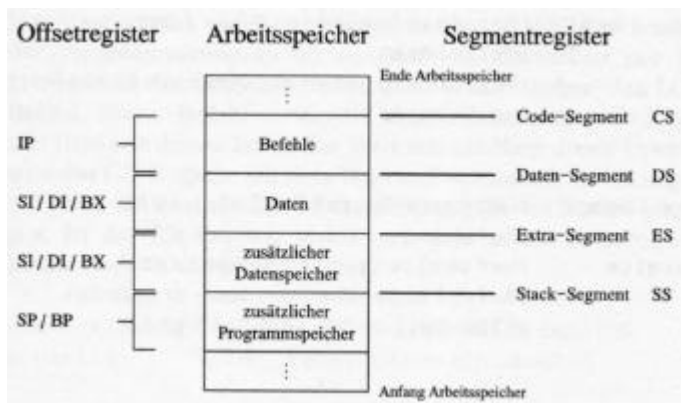
4.2 Segmentregister und Adressierung des Arbeitsspeichers

Adressierung des Arbeitsspeichers

Um die Arbeitsweise der Segmentregister zu verstehen, müssen wir uns mit der Adressierung des Arbeitsspeichers befassen. Wie schon erwähnt, holt sich der Prozessor Daten und Anweisungen aus dem Arbeitsspeicher. Diese Daten sagen ihm, was er als nächsten Schritt ausführen soll. Damit der Prozessor zwischen dem Anweisungsteil mit den eigentlichen Maschinenbefehlen und dem Datenteil mit den Variablen unterscheiden kann, werden die Daten in verschiedene Speicherbereiche getrennt. Diese Speicherbereiche werden in der EDV-Sprache als Segmente bezeichnet.

Maschinenbefehle werden im Speicherbereich *Code-Segment* gesammelt, die Variablen und Konstanten des Programms im Speicherbereich *Daten-Segment*. Wenn der Prozessor nun bestimmte Daten aus einem Segment lesen möchte, greift er dazu auf das dazugehörige Register zu.

Jedem Segment wird standardmäßig ein Register zugeordnet:



066.jpg

Da alle Register des 8086-Prozessors mit einem Wertebereich von 0 bis 65.535 nur eine Datenbreite von 16 Bit haben, können die Segmentregister nicht jede Stelle des ein Megabyte großen Arbeitsspeichers adressieren. Ein Megabyte entspricht der Zahl 1.048.576, nach der Binärlogik 2²⁰. Um die Speicherstellen von 0 bis 1.048.575 ansprechen zu können, benötigt man somit eine 20-Bit-Adresse. Zur Zeit der Konstruktion des Prozessors war es aber technisch nicht möglich, diesen mit 20 Bit breiten Registern zu bauen.

Um das Adressierungsproblem zu umgehen, werden zur Adressierung der Segmente im Arbeitsspeicher daher immer zwei Register benutzt. Das erste Register ist das Segmentregister, das zweite wird als Offsetregister bezeichnet.

Das Segmentregister zeigt auf den Startpunkt des Speicherbereiches im Arbeitsspeicher. Dieser Startpunkt des Speicherbereiches kann nicht willkürlich gewählt werden.

Da die Segmentregister nur 16 Bit breit sind, kann ein Segmentregister nur 65.536 Bytes adressieren. Um den ganzen Arbeitsspeicher adressieren zu können, ersann man folgenden Trick:

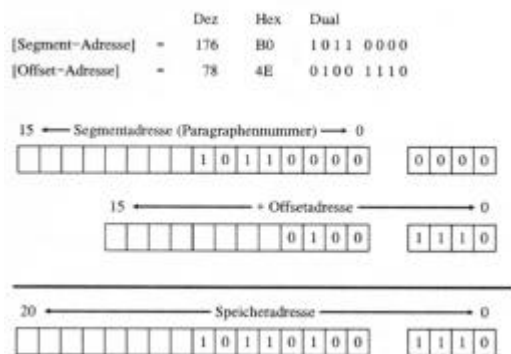
Man teilte den ein Megabyte großen Arbeitsspeicher durch 16, was 65.536 verschiedene Punkte im Arbeitsspeicher ergibt, die in einem Abstand von 16 Byte zueinander liegen. Diese Punkte im Arbeitsspeicher heißen Paragraphen.

Benötigt man nun eine 20-Bit-Adresse, um einen Paragraphen im Arbeitsspeicher adressieren zu können, so muss der jeweilige

Inhalt des Segmentregisters mit dem Wert 16 multipliziert werden. Dies ist in der hexadezimalen Rechenweise sehr einfach, da immer nur die Ziffer 0 angehängt wird. Aus dem 16-Bit-Segmentregister, mit dem möglichen hexadezimalen Inhalt 100h, wird durch Anhängen einer 0 die gewünschte 20-Bit-Größe 1000h erzeugt. Im Prozessor wird dazu der Inhalt des jeweiligen Segmentregisters in den Adressrechner gestellt und um 4 Bits nach links geschoben. Das entspricht einer Multiplikation im Dualsystem mit dem Wert 16. Diese 20-Bit-Adresse gibt nun den Startpunkt des jeweiligen Segmentes an. Um sich innerhalb des Segmentes bewegen zu können, wird ein zweites Register benötigt, das so genannte Offsetregister. Der Abstand vom Startpunkt des Segmentes zu der Speicherstelle, die im Segment adressiert werden soll, befindet sich im Offsetregister.

Segment- und Offsetregister bilden die erforderliche 20-Bit-Größe. Sie ist notwendig, um den Arbeitsspeicher zu adressieren. Da das Offsetregister eine Breite von 16 Bit hat, sind 64 KByte der größtmögliche Speicherplatz, den eine Variable in einem Programm einnehmen kann.

Bildung einer physikalischen Arbeitsspeicheradresse aus Segment- und Offsetadresse im Adressrechner. In diesem Beispiel beginnt das Segment an der Adresse 176 im Arbeitsspeicher. Die benötigte Offsetadresse soll 78 Byte vom Beginn des Segmentes entfernt sein.



068.jpg

Physikalische Arbeitsspeicheradresse

Segmentadresse mit 16 multiplizieren (im Adressrechner um 4 Bits nach links schieben) + Offsetadresse.

Die Segmentregister

In einem Programm können beliebig viele Segmente definiert werden. Mit den Segmentregistern CS, DS, ES und SS können gleichzeitig jedoch nur vier davon angesprochen werden. Möchten Sie auf Daten anderer Segmente zugreifen, so müssen die Segmentregister umgeladen werden. Dies kommt in der Praxis selten vor, da Daten und Code jeweils selten mehr als 64 Kilobyte einnehmen. Eine Ausnahme ist die Verwendung von externen Unterprogrammen. Hierbei werden das Daten- und das Code-Segment standardmäßig neu geladen.

In der Regel sind die einzelnen Segmente erheblich kleiner als 64 Kilobyte. In diesem Fall werden die einzelnen Segmente an der nächsten freien Paragraphennummer des vorhergehenden Segments angeschlossen. Schauen wir uns die vier Segmentregister des 8086 näher an.

Das CS-Register

Das CS-Register enthält die Anfangsadresse des maximal 64 Kilobyte großen Codesegments. Im Codesegment befinden sich die Maschinenbefehle, die beim HOLEN-Zyklus vom Prozessor gelesen werden. In Verbindung mit dem Befehlszeiger (Instruction-Pointer) als Offsetregister bildet der 8086 die 20-Bit-Adresse (CS:IP), um die Maschinenbefehle adressieren zu können.

Das DS-Register

Die Anfangsadresse des Datensegmentes befindet sich im DS-Register. Das Datensegment ist ein maximal 64 Kbyte großer Speicherbereich, in dem Daten in Form von Variablen, Feldern und Tabellen eines Programms angelegt werden. Jeder Maschinenbefehl, der diese Daten anfordert, greift standardmäßig auf das DS-Register zu. Um die physikalische 20-Bit-Adresse zu bilden, können vom DS-Register die drei Offsetregister SI, DI und BX verwendet werden.

Das ES-Register

Reicht der Platz im Datensegment nicht aus, so kann das Extrasegment als zusätzlicher Datenspeicher für die Aufnahme von Variablen verwendet werden. Das ES-Register zeigt dabei auf den Anfang des Extrasegmentes. Dieser maximal 64 Kilobyte große Speicherbereich wird besonders beim Kopieren und Verschieben von Zeichenketten verwendet. Alle String-Befehle zum Kopieren, Vergleichen und Durchsuchen von Zeichenblöcken verwenden standardmäßig das ES-Register. Die zum Basisregister ES zugehörigen Offsetregister stellen sich aus den Registern SI, DI und BX zusammen.

Das SS-Register

Das Stacksegment wird vom Programm als interner Speicher verwendet. Der Stack oder Stapel bezeichnet den maximal 64 Kilobyte großen Arbeitsspeicherbereich, der für die Zwischenspeicherung von Daten während des Programmlaufs verwendet wird. Dieser Speicherbereich kann vom Programm und vom Programmierer genutzt werden. Das Programm verwendet den Stack zum kurzzeitigen Zwischenspeichern der

Rücksprungadressen bei Unterprogrammaufrufen. Der Programmierer verwendet den Stack zum schnellen Zwischenspeichern von Daten und zur Parameterübergabe in Verbindung mit Hochsprachenprogrammen. Das SS-Register zeigt auf den Anfang des Stacksegments. Das zum Stacksegment nötige Offsetregister ist der Stackpointer SP. Der Stackpointer zeigt immer auf die aktuelle Position im Stacksegment. Die Adresse im Stack wird durch das Registerpaar SS:SP gebildet.

4.3 Zeige- und Indexregister

Die Zeige- und Indexregister unterstützen die Adressierung in den vorgestellten Segmenten. Diese Register beinhalten die Länge vom Startpunkt des Segmentes bis zu der Speicherstelle, die im Segment adressiert werden soll. Diese Länge wird als Offset bezeichnet.

Der Befehlszeiger IP

Der Befehlszeiger, als Instruction-Pointer oder auch als Programm-Counter bezeichnet, adressiert in Verbindung mit dem Codesegment den nächsten auszuführenden Maschinenbefehl. Wird das Programm der Reihe nach Befehl für Befehl abgearbeitet, so enthält der Befehlszeiger den Offset der Speicheradresse, die den nächsten auszuführenden Maschinenbefehl beinhaltet.

Mit dem Codesegment zusammen ergibt das Registerpaar CS:IP eine 20-Bit-Adresse, deren Inhalt von der BIU beim nächsten HOLEN-Zyklus in das Befehlsregister geladen wird. Nach dem HOLEN-Zyklus wird die Länge des eingelesenen Befehls in den Befehlszeiger addiert. Der Befehlszeiger zeigt damit wieder auf den nächsten im Codesegment befindlichen Maschinenbefehl. Dies gilt aber nur für die hier beschriebene sequentielle Abarbeitung des Programms.

Unterbricht ein Programm die sequentielle Befehlsabarbeitung durch den Aufruf eines Unterprogramms oder eines Sprungbefehls, so wird die weitere Abarbeitung des Programms an einer anderen Stelle im Arbeitsspeicher fortgeführt. Ein Unterprogrammaufruf oder ein Sprungbefehl ist also nichts anderes als das Laden einer anderen Speicheradresse in den Befehlszeiger. Zusätzlich wird bei jedem Sprung oder Unterprogrammaufruf die Befehlswarteschlange gelöscht, da die durch die BIU vorgelesenen Befehle für die weitere Verarbeitung nicht mehr verwertbar sind.

Liegt der Sprung oder Unterprogrammaufruf im aktuellen Codesegment, wobei das maximal 64 Kilobyte große Codesegment nicht verlassen wird, so spricht man von einem *Near-Jump* bei einem Sprung oder von einem *Near-Call* bei einem Unterprogrammaufruf. Überschreitet der Sprungbefehl oder der Unterprogrammaufruf die Grenzen des aktuellen Datensegments, so muss zum Befehlszeiger das Datensegment mit einer neuen Adresse zusätzlich geladen werden. In diesem Fall spricht man von einem *Far-Jump* oder von einem *Far-Call*.

Der Befehlszeiger kann im Gegensatz zu den anderen Registern nicht vom Programmierer angesprochen werden. Nur durch die Sprungbefehle und Unterprogrammaufrufe kann der Befehlszeiger verändert werden.

Die Indexregister SI und DI

Die beiden Indexregister mit den Namen SI (Source-Index / Quell-Index) und DI (Destination-Index / Ziel-Index) unterstützen als Offsetregister die Adressierung im Datensegment und Extrasegment.

Der Basepointer (BP)

Will man im Stacksegment auf zwischengespeicherte Werte oder Parameter zugreifen, so wird der Basepointer (BP) benötigt. Wie die anderen Zeige- und Indexregister BX, SI und DI kann auch BP als Index verwendet werden.

Der Stackpointer (SP)

Der Stackpointer zeigt stets auf die aktuelle Position im Stacksegment. Die physikalische Adresse zum Zugriff auf das Stacksegment wird durch das Registerpaar SS:SP dargestellt.

Obwohl der Stackpointer, wie die Allzweckregister, für die meisten arithmetischen und logischen Befehle als 16-Bit-Register verwendet werden kann, sollten Sie das mit dem Stackpointer vorsichtshalber nicht tun. Denn wenn Sie den Stackpointer verändern, zeigt dieser nicht mehr auf die aktuelle Position im Stacksegment.

Wenn Sie Unterprogramme aufrufen, so speichert das System den Inhalt des Befehlszeigers auf dem Stack ab. Das System benötigt diesen Wert, um nach der Abarbeitung des Unterprogramms wieder an der richtigen Speicherstelle im Programm weiterarbeiten zu können. Diese Speicherstelle ist der nächste Befehl nach dem Unterprogrammaufruf.

Sie erinnern sich: Nach dem Einlesen des nächsten Maschinenbefehls durch die BIU wird die Länge des auszuführenden Befehls auf den Befehlszeiger aufaddiert. Der Befehlszeiger zeigt damit schon auf den nächsten im Codesegment befindlichen Maschinenbefehl. Dieser wird vor dem Unterprogrammaufruf auf dem Stack zwischengespeichert. Auch viele der Systemfunktionen setzen den Stack für ihre Anliegen ein. Man weiß also nie, wer den Stack gerade benutzt.

Wenn Sie das Stacksegment durch den Stackpointer verändern, ist der richtige Zustand nicht vorhanden, wenn eine Systemfunktion ihn benötigt, und das kann sehr schnell zu einem Disaster führen.

Da wir die ganze Zeit über den Stack reden, diesen aber noch gar nicht richtig erklärt haben, wollen wir dies sofort nachholen.

4.4 Stack

Der Stack oder Stapel ist ein Zwischenspeicher, mit dem der Prozessor und der Programmierer kurzzeitig Daten Zwischenspeichern können. Den Namen Stapel verdient das Stacksegment dabei zu Recht, da die zwischengespeicherten Daten nach dem *Last in - First out* (LIFO) Prinzip abgelegt und wieder abgeholt werden. Dazu können wir uns einen Stapel Teller

vorstellen. Möchten Sie einen Wert aus einem Register sichern, legen Sie den Registerinhalt wie einen Teller auf den Stapel. Der Stackpointer wird um den Wert 2 verringert und zeigt wieder auf die Spitze des Stapels. Man spricht dabei auch vom *Pushen* eines Wertes auf dem Stack, der dazugehörige Befehl, um den Inhalt eines Registers auf dem Stack zu sichern, heißt *Push*. Wenn der Wert vom Stapel geholt werden soll, muss dieser wie ein Teller von einem Stapel heruntergenommen werden. Dies bezeichnet man als *Poppen* eines Wertes, wofür es in der Programmiersprache Assembler den Befehl *Pop* gibt.

Beispiel

Sie verändern in einem Unterprogramm die Register AX und BX, wissen aber nicht, ob in diesen Registern wichtige Daten stehen. Abhilfe schafft hier das kurzzeitige Zwischenspeichern der Werte aus den Registern auf dem Stack. Mit der folgenden Befehlsfolge gehen Sie ganz sicher:

```

mov ax, wichtiger_wert ; Register AX mit dem Inhalt der Variable wichtiger_wert laden
mov bx, noch_wichtiger ; Register BX mit dem Inhalt der Variable noch,, wichtiger laden
call Upro              ; Unterprogramm mit dem Namen Upro aufrufen
...
Upro PROC NEAR        ; Anfang des Unterprogramms
  push ax             ; Register AX auf den Stack retten
  push bx            ; Register BX auf den Stack retten
  ...
; <Befehle ausführen und die Register AX und BX verändern>
  ...
  pop bx             ; Register BX wieder vom Stack holen
  pop ax            ; Register AX wieder vom Stack holen
  ret               ; Rücksprung zum Aufruf Ende des Unterprogramms
Upro ENDP

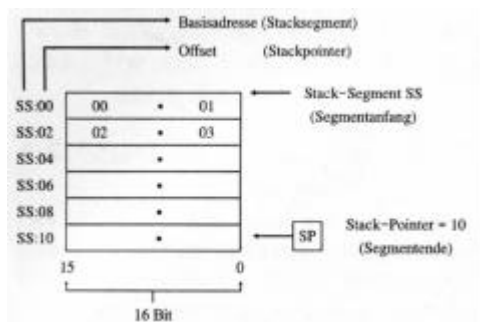
```

Befehl *Push* können Sie alle 16-Bit-Register, mit Ausnahme des Befehlszeigers, auf dem Stack Zwischenspeichern. Das Stacksegment kann immer nur wortweise (2 Byte) beschrieben (Push) und wortweise gelesen (Pop) werden. Der Stack ist durch diese Datenbreite von 2 Byte für das kurzzeitige Zwischenspeichern von Registerinhalten (16 Bit) prädestiniert. Für diese Zwecke wird der Stack vorwiegend verwendet:

- für das „Retten“ von Registerinhalten durch den Programmierer,
- für die Unterprogrammtechnik.

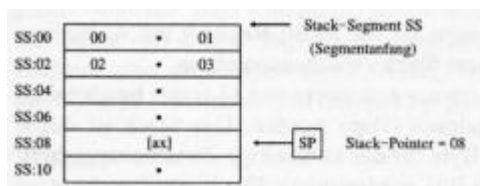
Beim Ablegen eines 16-Bit-Wertes auf dem Stack (Push) wird der Stackpointer um den Wert 2 verringert. Wenn der 16-Bit-Wert umgekehrt vom Stack genommen wird (Pop), wird der Stackpointer um den Wert 2 vergrößert. Der Stack wächst also von den hohen zu den niedrigen Adressen. Das nachfolgende Bild soll dies verdeutlichen.

Anfangszustand: Der Stackpointer steht am Ende des Stacksegments, welches in diesem Beispiel eine Größe von 10 Byte hat.



074a.jpg

Nachfolgende Abbildung zeigt den Stack nach Ausführung des Befehls *push ax*, der den Inhalt des AX-Registers (16 Bit) auf dem Stack zwischengespeichert hat.

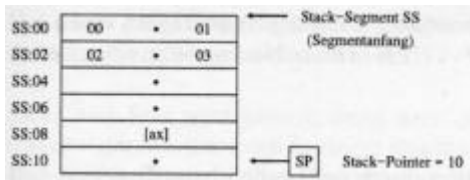


074b.jpg

Beim Push-Befehl wird zuerst der Stackpointer um 2 reduziert ($SP - 2$) und dann der Wert der beim Push-Befehl angegebenen Operanden auf den Stack gespeichert (Wert \rightarrow [SS:SP]).

`pop ax` ; Hole den Inhalt des obersten Stackelements in das Register AX

Der Stack nach Durchführung des Befehls `pop ax`, der den 16-Bit-Wert vom Stack in das AX-Register kopiert:



075a.jpg

Beim Befehl *Pop* wird zuerst der Wert auf den Stack ([SS:SP] \rightarrow Wert) des beim Push-Befehl angegebenen Operanden kopiert und dann der Stackpointer um 2 erhöht ($SP + 2$). Durch die Befehle *Push* oder *Pop* kann der Stack wortweise gelesen oder beschrieben werden.

4.5 Statusregister

Das Statusregister, auch Flagregister oder Programm-Status-Wort genannt, zeigt Informationen über das Ergebnis eines Maschinenbefehls an. Das Statusregister ist wie seine Kollegen ein 16-Bit-Register. Im Unterschied zu diesem wird der Inhalt des Statusregisters über einzelne Bits angezeigt. Diese Bits, die auch Flags genannt werden, sind dabei folgendermaßen angeordnet:

Der Aufbau des Statusregisters



075b.jpg

Ein Flag im Statusregister ist ein Bit, das einen bestimmten Zustand anzeigt. Steht dieses Bit auf 1, so trifft der Zustand, den das Flag vertritt, zu. Steht es auf 0, so trifft der Zustand nicht zu. Führen Sie beispielsweise in einem Programm eine Subtraktion mit dem Ergebnis null durch, so wird das Zero Flag auf 1 gesetzt.

Dieser Zustand im Statusregister kann von Ihnen durch einen bedingten Sprung wie ein IF-THEN in einer Hochsprache ausgewertet werden.

Beispiel

Sie führen eine Subtraktion durch und wollen beim Ergebnis null eine Entscheidung treffen. Der durch das Zero Flag gesteuerte bedingte Sprung heißt in der Programmiersprache Assembler *jump on zero = jz* (springe wenn null) oder die Verneinung *jump not zero = jnz* (springe wenn nicht null).

```

...
mov ah, 7           ; Schiebe den Wert 7 in das Register AH
sub ah, 7           ; Ziehe vom Inhalt des Registers AH den Wert 7 ab
jnz Technik        ; Springe bei nicht null zu dem Label Technik:
...
; <Weiter geht' s>

```

```

Technik:
...
; <Prozessor austauschen>
...

```

Von den vorhandenen 16 Bits im Statusregister werden beim 8086-Prozessor nur 9 Bits (Flags) benötigt. Diese 9 Flags unterteilen sich dabei in 6 Statusflags und 3 Kontrollflags. Die restlichen Bits werden von den Nachfolgeprozessoren des 8086 benutzt, dies spielt aber nur in Multitaskingbetriebssystemen wie OS/2 eine Rolle.

Die Statusflags

Die 6 Statusflags werden vom Prozessor gesetzt, um den Programmierer über bestimmte Ereignisse zu informieren. Alle Statusflags werden vor jedem Befehl auf 0 gesetzt und damit gelöscht. Sie können durch bedingte Sprungbefehle im Programm ausgewertet werden.

Das Carry Flag/Übertragflag (CF), BitNr. 0

CF = 1 = Wertebereich überschritten.

Das Carry Flag wird gesetzt, wenn nach einer Addition oder Subtraktion der Wertebereich in einem Register überschritten ist. Dieses Flag zeigt somit den Übertrag aus dem höchstwertigen Bit an. Findet ein Übertrag statt, so steht das Carry Flag auf 1. So kann beispielsweise bei einer Addition zweier 8-Bit-Werte das Resultat einen 9-Bit-Wert darstellen. In diesem Fall kann das Ergebnis von einem 8-Bit-Register nicht mehr aufgenommen werden. Das Carry Flag stünde somit auf eins. Das Carry Flag wird also immer dann gesetzt, wenn bei einer Addition oder Subtraktion ein Übertrag beim höchstwertigen Bit erfolgt.

Beispiel

Die Addition zweier vorzeichenloser 8-Bit-Zahlen.

(Dual)	(Dezimal)
1 1 1 1 0 0 0 0	240
+ 0 0 0 1 0 1 0 0	+ 20
1 1 1	<- Übertrag
1 0 0 0 0 0 1 0 0	260

Mit 8 Bit lassen sich Werte von 0 bis 255 darstellen. Das Ergebnis der Addition wäre im obigen Beispiel durch ein 8-Bit-Register nicht mehr darstellbar. Um diesen Fehler anzuzeigen, wird von der CPU das Carry Flag auf 1 gesetzt. In einem Programm müssten wir diesen möglichen Fehler durch einen bedingten Sprung, der sich auf das Carry Flag bezieht, berücksichtigen.

Beispiel

Der durch das Carry Flag gesteuerte bedingte Sprung heißt in der Programmiersprache Assembler `jump on carry = je` (springe, wenn Carry Flag gesetzt).

```

...
mov ah, 240          ; Schiebe den Wert 240 in das AH-Register
add ah, 20           ; Addiere zu dem Inhalt im AH-Register den Wert 20
jc Ueberlauf        ; Springe bei Fehler zu dem Label Ueberlauf:
...
; <alles ok>
...
Ueberlauf:
...
; <Fehlerbehandlung>
...

```

Nach dieser Addition ist das Carry Flag gesetzt, so dass Sie das Überschreiten des Wertebereichs von 256 erkennen und entsprechend darauf reagieren können.

Das Carry Flag kann vom Programmierer durch den Befehl `CLC` gelöscht (CF = 0) und durch `STC` gesetzt werden (CF = 1). Dadurch kann es auch als Schalter verwendet werden.

Das Parity Flag / Paritätsflag (PF), BitNr. 2

PF = 1 = Ergebnis einer Operation enthält in den niederwertigen 8 Bits eine gerade Anzahl an gesetzten Bits.

Dieses Flag dient zur Fehlerprüfung bei der Datenübertragung über die serielle Schnittstelle. Das Parity Flag ist auf 1, wenn das Ergebnis der beiden Prüfsummen eine gerade Anzahl an gesetzten Bits im Low-Byte ergibt. Bei einer ungeraden Anzahl von gesetzten Bits wird das Parity Flag auf 0 gesetzt.

Beispiel

	(Dual)	(Dezimal)
	0 1 0 0 0 1 0 0	68
+	0 1 1 0 0 1 0 1	+ 101
	-----	-----
	1 0 1 0 1 0 0 1	169

Die Anzahl der gesetzten Bits im Low-Byte ist gerade (PF = 1)

	(Dual)	(Dezimal)
	0 1 0 0 0 1 1 0	70
+	0 1 1 0 0 1 0 1	+ 101
	-----	-----
	1 0 1 0 1 0 1 1	171

Die Anzahl der gesetzten Bits im Low-Byte ist ungerade (PF = 0)

Das Auxiliary Flag / Hilfsübertragsflag (AF), BitNr. 4

AF = 1 = Übertrag aus Bit 3 nach Bit 4 bei 8-Bit-Operation.

Dieses Flag ist dem Carry Flag sehr ähnlich. Zum Unterschied, wird mit dem Hilfsübertragsflag ein Übertrag von Bit 3 nach Bit 4 bei einer 8-Bit-Operation angezeigt. Bei einem Übertrag steht AF auf 1. Das Hilfsübertragsflag wird hauptsächlich in der BCD-Arithmetik verwendet.

Wie im Kapitel 2 bereits erwähnt, werden Zahlen in der BCD-Darstellung in eine eigene 4-Bit-Zahl umgewandelt. Die Zahl 29 wird in der BCD-Darstellung dual mit 0010 1001 dargestellt, die Zahl 38 mit 0011 1000 usw. Die Ziffer einer Dezimalzahl kann in der BCD-Darstellung nur Werte von 0 bis 9 einnehmen. Bei der nachfolgenden Addition der Zahlen im BCD-Format würde also folgender Fehler auftreten:

	(BCD-Format)	(Dezimal)	
	0 0 1 0 1 0 0 1	29	
+	0 0 1 1 1 0 0 0	+ 38	
	1		<- Übertrag
	-----	-----	
	0 1 1 0 0 0 0 1	67	

Das AF Flag meldet in diesem Fall, dass die Addition der BCD-Zahlen fehlerhaft war. Mit dem Befehl AAA könnte das Ergebnis anschließend wieder in ein korrektes BCD-Format gebracht werden.

Das Zero Flag / Nullflag (ZF), BitNr. 6

ZF = 1 = Ergebnis einer Operation ist null.

Das Zero Flag zeigt nach einer Operation an, ob das Ergebnis null ist oder nicht. Beim Ergebnis 0 steht das Zero Flag auf 1. Dieses Flag wird mit dem Schleifenbefehl *Loop* eingesetzt, bei Vergleichsbefehlen und allen arithmetischen Operationen. Bei einem Vergleich zweier Zahlen werden diese intern subtrahiert, ohne die jeweiligen Register zu verändern. Ist das Ergebnis der Subtraktion null, so sind beide Zahlen gleich groß, und das Zero Flag wird gesetzt.

Beispiel

Vergleiche zwei Operanden miteinander, und werte das Ergebnis aus. Der Befehl, um zwei Operanden miteinander zu vergleichen, heißt in der Programmiersprache Assembler *Compare = cmp*, der bedingte Sprung dazu *jump equal = je* (springe wenn gleich), die Verneinung *jump not equal = jne* (springe wenn nicht gleich).

```

...
mov ax, 222           ; Schiebe den Wert 222 in das AX- Register
cmp ax, 222          ; Vergleiche den Inhalt des AX- Registers mit dem Wert 222
je Gleich             ; Sind die beiden Operanden des CMP-Befehls gleich - Springe zu dem Label Gleich:
...

```

; <Operanden sind verschieden>

...

Gleich: .

; <Operanden sind gleich groß>

...

	(Dual)	(Dezimal)
	1 1 0 1 1 1 1 0	222
-	1 1 0 1 1 1 1 0	- 222
-----		-----
	0 0 0 0 0 0 0 0	0

Das Ergebnis der Subtraktion ist null. Das Zero Flag ist 1.

Das Sign Flag / Vorzeichenflag (SF), BitNr. 7

SF = 1 = Ergebnis ist negativ.

SF = 0 = Ergebnis ist positiv.

Das höchstwertige Bit einer Zahl wird zur Darstellung des Vorzeichens verwendet. Nach einem arithmetischen oder logischen Befehl befindet sich das höchstwertige Bit des Ergebnisses im Sign Flag. Das Sign Flag zeigt somit an, ob das Ergebnis positiv oder negativ ist. Ist das Ergebnis negativ, so wird das Sign Flag auf 1 gesetzt. Bei einem positiven Ergebnis oder bei null wird das Vorzeichenflag auf 0 gesetzt.

Beispiel:

	(Dual)	(Dezimal)
	0 0 1 1 0 0 0 1	49
+	0 0 0 1 0 1 0 0	+ 20
-----		-----
	0 1 0 0 0 1 0 1	69

Das Vorzeichen im Ergebnis ist positiv (höchstwertiges Bit = 0), das Sign Flag steht somit auf 0.

	(Dual)	(Dezimal)
	0 1 1 1 1 0 1 1	123
+	0 1 0 0 0 0 0 0	+ 64
-----		-----
	1 0 1 1 1 0 1 1	187

Bei der obigen Addition zweier positiver Werte mit Vorzeichen wurde der Wertebereich überschritten. Das Ergebnis wäre bei Werten

ohne Vorzeichen zwar richtig. Da wir in unserem Beispiel aber mit dem Vorzeichen rechnen, ist das Ergebnis negativ. Das Sign Flag steht somit auf 1. Um derartige Fehlersituationen abzufangen, gibt es zum Glück noch das Overflow Flag.

Das Overflow Flag / Überlaufflag (OF), BitNr. 11

OF = 1 = Vorzeichenbit zerstört.

Wenn bei einer arithmetischen Operation ein Übertrag auf das höchstwertige Bit erfolgt, wird das Overflow Flag auf 1 gesetzt. Dies ist nur bei Werten mit Vorzeichen von Bedeutung. Das Overflow Flag ist dabei nicht mit dem Carry Flag zu verwechseln. Das Carry Flag wird gesetzt, wenn der Übertrag aus dem höchstwertigen Bit eines Wertes verloren geht. Dies ist immer dann der Fall, wenn der Wertebereich überschritten wird. Ein 16-Bit-Wert beispielsweise, der die Bits 0 bis 15 zur Verfügung stellt, würde einen Übertrag vom Bit 15 in das nicht mehr vorhandene Bit 16 mit dem Carry Flag auf 1 quittieren. Das Overflow Flag würde den Übertrag vom Bit 14 in das Bit 15 anzeigen, da das Bit 15 bei vorzeichenbehafteten 16-Bit-Werten das Vorzeichen repräsentiert. Das Overflow Flag wird also gesetzt, wenn das Vorzeichenbit durch einen Übertrag verändert wird.

Beispiel

	(Dual)	(Dezimal)
	0 1 0 1 0 1 1 1	87
+	0 0 1 1 0 0 1 0	+ 50
-----		-----
	1 0 0 0 1 0 0 1	137

Bei der obigen Addition zweier positiver Werte mit Vorzeichen wurde der Wertebereich überschritten. Das Ergebnis wäre bei Werten ohne Vorzeichen zwar richtig, aber in unserem Beispiel ist das Ergebnis unter Berücksichtigung des Vorzeichens negativ.

Die Kontrollflags

Wir kommen nun zu den letzten drei Flags des Statusregisters. Im Gegensatz zu den vorgestellten Statusflags werden die Kontrollflags nicht von der CPU, sondern vom Programmierer gesetzt.

Das Trap Flag / Einzelschrittflag (TF), BitNr. 8

TF = 1 = Nach jedem Befehl rufe den Interrupt 1 auf, und verzweige in die dazugehörige Unterbrechungsroutine.

Das Trap Flag versetzt den Prozessor in den Einzelschrittmodus. Durch das Trap Flag wird der Prozessor angewiesen, nach jedem Befehl im Programm den Interrupt 1 aufzurufen. Was ein Interrupt ist, werden wir uns im Kapitel *Systemprogrammierung mit Interrupts* später noch ganz genau anschauen. Durch den Aufruf dieses Interrupts bleibt die CPU nach jedem ausgeführten Befehl stehen und verzweigt in eine spezielle Routine, wie beispielsweise einen Debugger. Ein Debugger ist ein Programm, das bei der Fehlersuche eingesetzt wird. Mit Hilfe des Debuggers kann der Programmierer jeden Schritt des Programms verfolgen und sehen, wie sich die einzelnen Register, Variablen und Flags verändern. Dies ist besonders bei der Fehlersuche in einem Programm sehr wichtig. MSDOS stellt dazu das Dienstprogramm *DEBUG* zur Verfügung. Wer den MASM oder TASM besitzt, sollte auf den Debugger in seiner Entwicklungsumgebung zurückgreifen, da er ein Vielfaches an Komfort gegenüber dem MS-DOS-Dienstprogramm *DEBUG* bietet.

Das Interrupt Enable Flag / Unterbrechungsflag (IF), BitNr. 9

IF = 1 = Alle Unterbrechungen zugelassen.

IF = 0 = Keine maskierbaren Unterbrechungen zugelassen.

Der Prozessor kann durch äußere Einflüsse bei der Abarbeitung eines Programms unterbrochen werden. Diese Unterbrechungen werden Interrupts genannt. Ein Interrupt kann von einem Peripheriegerät wie der Tastatur ausgelöst werden. Viele Programme können beispielsweise durch die Tastenkombination Strg + C abgebrochen werden. Wenn Sie aber während der Verarbeitung des Programms eine solche Unterbrechung nicht wünschen, so können Sie dies durch Löschen des Unterbrechungsflags verhindern. Gesetzt wird das Interrupt Enable Flag mit dem Befehl *STI*, gelöscht wird es mit *CLI*.

Das Direction Flag / Richtungsflag (DF), BitNr. 10

DF = 1 = Stringverarbeitung nach aufsteigenden Adressen (SI und DI werden erhöht).

DF = 0 = Stringverarbeitung nach absteigenden Adressen (SI und DI werden verringert).

Bei der Stringverarbeitung werden die zu bearbeitenden Speicherblöcke immer nach aufsteigenden Adressen von links nach rechts bearbeitet. Dazu werden die Register DI oder SI nach jedem durchlaufenem Byte (Zeichen) erhöht. Möchten Sie die Zeichenketten jedoch nach absteigenden Adressen von rechts nach links verarbeiten, so müssen Sie dies dem Prozessor über das Richtungsflag mit dem Befehl *STD* mitteilen. Um wieder mit aufsteigenden Adressen arbeiten zu können, ist das Richtungsflag mit *CLD* zu löschen.

4.6 Speicheroperanden adressieren

Immer wenn die BIU Daten aus dem Arbeitsspeicher holen oder in den Arbeitsspeicher zurücktransportieren will, muss dazu eine 20-Bit-Adresse aus dem Segmentregister und dem Offsetregister gebildet werden. Das Segmentregister zeigt dabei auf den Anfang des Segmentes. Mit dem Offsetregister kann man sich dann innerhalb des 64 Kilobyte großen Segmentes bewegen, um die Operanden im Datensegment anzusprechen. Der Inhalt des Offsetregisters besteht beispielsweise aus der Angabe einer einfachen Offsetadresse, wie die Angabe einer Variablen, oder aus einem Verbund der Indexregister, aus denen die Offsetadresse erst errechnet werden muss. Diese können wiederum miteinander kombiniert werden.

Um dieses komplexe Thema etwas zu konkretisieren, sei gleich ein Beispiel gegeben:

Stellen Sie sich vor, Sie vereinbaren im Programm zwei Variablen mit den Namen *Max* und *Moritz*, die jeweils eine Größe von 2 Bytes besitzen. Folgende Befehle sind dazu notwendig:

```
Max    dw    0           ; <---- steht im Codesegment an Speicherstelle 100
Moritz dw    0           ; <---- steht im Codesegment an Speicherstelle 102
```

Um den Inhalt der Variablen *Moritz* in das AX-Register zu bekommen, ist folgender Befehl denkbar:

```
...
mov   ax, Moritz       ; Schiebe den Inhalt der Variablen Moritz in das AX-Register
...
```

Nach dem Übersetzen des Programms durch den Assembler wird der Name der Variablen *Moritz* in eine Adresse umgewandelt:

```
...
mov ax, [102]           ; Schiebe den Inhalt der Speicherstelle 102 in das AX-Register
...
```

Die Adressangabe 102 ist dabei unsere Offsetadresse im Datensegment. Sie wird auch als effektive Adresse bezeichnet. Die effektive Adresse könnte man auch anders bilden.

Um den Inhalt der Variablen *Moritz* in das AX-Register zu kopieren, sind folgende weiteren Befehle denkbar:

```
...
mov bx, offset Max     ; Schiebe die Adresse der Speicherstelle, an der die Variable Max
                        ; definiert ist, in das BX-Register
...
mov si, 2              ; Schiebe den Wert 2 in das SI-Register
mov ax, [bx + si]      ; Schiebe den Inhalt der durch das Registerpaar BX und SI
                        ; adressierten Speicherstelle in das AX-Register
...
```

Während der Laufzeit des Programms wird folgende Adressberechnung vorgenommen:

```
...
mov bx, 100            ; Schiebe die Adresse der Speicherstelle, an der die Variable Max
                        ; definiert ist, in das BX-Register
...
mov si, 2              ; Schiebe den Wert 2 in das SI-Register
mov ax, [100 + 2]      ; Schiebe den Inhalt der Speicherstelle 100 + 2 in das AX-Register
...
```

Die im letzten Beispiel behandelte Adressierungsmethode ist in diesem Fall natürlich Unsinn. Diese Adressierungsart eignet sich besonders für die Abarbeitung mehrdimensionaler Tabellen.

Die Offsetadresse oder effektive Adresse muss nur berechnet werden, wenn ein Zugriff auf den Arbeitsspeicher benötigt wird.

Wenn der zu verarbeitende Wert direkt im Register steht oder der Wert direkt im Befehl mit angegeben ist, entfällt die Berechnung der Offsetadresse. Die CPU muss den Operanden nicht aus dem Arbeitsspeicher anfordern.

Die Offsetadresse oder effektive Adresse kann beim 8086-Prozessor durch fünf verschiedene Adressierungsarten gebildet werden. Alle vorgestellten Adressierungsarten werden in den folgenden Kapiteln noch weiter vertieft. Die Adressierungsarten lassen sich in zwei Gruppen aufteilen. Die Aufteilung erfolgt in die Befehlsadressierung, in der sich Operanden als Direktwert oder als Registerinhalt direkt im Befehl selbst befinden, und die Arbeitsspeicheradressierung, in der die Operanden erst über den Adress- und Datenbus aus dem Speicher geholt werden müssen.

Bei den beiden nachfolgenden Adressierungsarten handelt es sich um Operanden, die direkt im Maschinenbefehl mit angegeben werden. Diese Operanden werden als Bestandteile des Maschinenbefehls geladen. Zusätzliche Zugriffe auf den Arbeitsspeicher werden dadurch nicht benötigt. Das erhöht die Geschwindigkeit des Programms sehr.

1. Die Direktwertadressierung

```
mov ax, 5000           ; Schiebe den Wert 5000 in das AX-Register
```

Bei der Direktadressierung wird ein Register oder eine Variable mit einem 8- oder 16-Bit-Wert (Operand) direkt geladen. Der Direktwert ist dabei ein fester Bestandteil des Maschinenbefehls und braucht deshalb nicht aus dem Arbeitsspeicher geladen zu werden.

2. Die Registeradressierung

```
add ax, bx             ;Addiere den Inhalt des BX-Registers auf den Inhalt des AX-Registers
```

Bei der Registeradressierung befindet sich der Operand in einem Register. Ein Zugriff auf den Arbeitsspeicher ist nicht erforderlich, da die angesprochenen Register im Maschinenbefehl selbst mit angegeben sind.

Bei den folgenden Adressierungsarten müssen die Operanden, die im Befehl angegeben sind, erst aus dem Datensegment im Arbeitsspeicher geholt werden. Dazu muss der BIU eine Länge übergeben werden, die den Abstand des gewünschten Operanden im Arbeitsspeicher zum Segmentanfang angibt. Diese Länge wird als Offset oder effektive Adresse bezeichnet. Wie diese effektive Adresse vom Programmierer gebildet werden kann, wollen wir uns nun näher anschauen.

3. Die direkte Adressierung

```
mov ax, Zaehler ; Schiebe den Inhalt der Variable Zaehler in das AX-Register oder genauer  
; lade den Inhalt der Speicherstelle mit der symbolischen Adresse Zaehler  
; aus dem Datensegment in das AX-Register .  
  
inc Zaehler ; Erhöhe den Inhalt der Variable Zaehler um den Wert 1
```

Bei der direkten Adressierung wird dem Prozessor die effektive Adresse einer Speicherstelle in Form einer Variablen angegeben. Diese Variable wird dann bei der Befehlsausführung aus dem Datensegment im Arbeitsspeicher in den Prozessor transportiert.

Als Basisregister wird standardmäßig das Datensegment (DS) verwendet. Möchten Sie eine Variable verwenden, die Sie im Extrasegment definiert haben, so müssen Sie dies in folgender Form angeben:

```
mov ax, es:Zaehler ; Schiebe den Inhalt der im Extrasegment definierten Variablen in das AX-Register
```

4. Die indirekte Registeradressierung

```
mov ax, [ bx ] ; Lade den Inhalt der Speicheradresse im BX-Register in das AX-Register  
mov [bp + di], ax ; Schiebe den Inhalt des AX-Registers in die durch die beiden Register BP  
; und DI adressierte Speicherstelle.  
; Addiere dazu den Inhalt der beiden Register BP und DI. Die resultierende  
; Summe ist die Adresse der zu beschreibenden Speicherstelle.
```

Hier befindet sich die Offsetadresse des Operanden in einem Basis oder Indexregister, im BX-, BP-, DI- oder SI-Register. Es wird also der Inhalt eines oder mehrerer dieser Register verwendet, um die Speicheradresse zu lokalisieren. Die beiden Basis- und Indexregister dürfen dabei nicht kombiniert werden. Es ergeben sich folgende Möglichkeiten der Adressierung:

- [bx]
- [bp]
- [di]
- [si]
- [bx + di] oder [di + bx]
- [bx + si] oder [si + bx]
- [bp + di] oder [di + bp]
- [bp + si] oder [si + bp]

5. Die indizierte Adressierung

```
mov [Tabelle + bx], bx ; Schiebe den Inhalt des BX-Registers in die Speicher-  
; stelle, die durch die symbolische Adresse Tabelle in  
; Verbindung mit der Länge aus dem BX-Register gebildet wird.
```

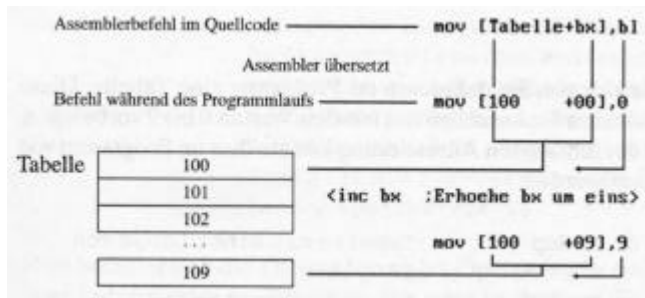
Wie der Name schon sagt, handelt es sich um die Adressierung durch einen Index. Diese Adressierung wird besonders bei Tabellen verwendet. Die effektive Adresse wird aus einer festen Adressangabe in Verbindung mit einem Index errechnet.

Beispiel

Stellen Sie sich vor. Sie definieren im Programm eine Tabelle. Diese Tabelle möchten Sie anschließend mit den Werten 0 bis 9 vorbelegen. Mit Hilfe der indizierten Adressierung könnte dies im Programm wie folgt kodiert werden:

```
Tabelle db 10 dup (?) ; Tabelle mit einer Länge von 10 Byte  
laenge equ $ - Tabelle ; Länge der Tabelle  
  
push bx ; Rette den Inhalt des BX-Register  
push cx ; Rette den Inhalt des CX- Register  
mov bx, 0 ; Schiebe den Wert 0 in das BX-Register. Tabelle + 0 ist der erste Eintrag  
mov cx, laenge ; CX = Länge der Tabelle = 10 = die Schleife wird 10 mal durchlaufen.  
  
Schleife:  
mov [Tabelle + bx] ; bl; Schiebe den Inhalt von BL in die symbolische Speicheradresse Tabelle + BX  
inc bx ; Inhalt in BX-Register um 1 erhöhen  
loop Schleife ; Solange CX-Register nicht null, springe zu dem Label Schleife:  
pop cx ; Alten Inhalt der Register CX und BX  
pop bx ; wiederherstellen
```

Wir gehen davon aus, dass die Variable *Tabelle* an der Speicherstelle 100 im Datensegment angelegt ist. Im Arbeitsspeicher stellt sich das Ganze dann so dar:



090.jpg

Adressierungsarten

Die effektive Adresse (eA) ergibt in Verbindung mit dem Segmentregister die benötigte 20-Bit-Adresse für den Zugriff auf den Arbeitsspeicher. Standardmäßig wird als Segmentregister das Datensegment (DS) verwendet. Ist jedoch das BP-Register (BP) an der Bildung der effektiven Adresse beteiligt, so wird die 20-Bit-Adresse mit dem Inhalt des Stacksegments (SS) gebildet (SS:BP). Mit einem Segmentregisterpräfix kann diese Zuordnung umgeleitet werden.

Beispiel

Das BP-Register soll mit dem Datensegment zusammenarbeiten.

Folgende Kodierung des Befehls wäre erforderlich:

```
mov ax, [ds:bp]      ; Zugeordnetes Segmentregister ist das Datensegment
mov ax, [bp]         ; Zugeordnetes Segmentregister ist das Stacksegment
```

Adressierungsart	Bildung der eA	zugeord.Segmentregister
Direkte Adressierung	Label	DS
	[Adresse]	DS
Indirekte Registeradressierung	[bx]	DS
	[bp]	SS
	[di]	DS
	[si]	DS
	[bx+di]	DS
	[bx+si]	DS
	[bp+di]	SS
[bp+si]	SS	
Indizierte Registeradressierung	[Label+bx]	DS
	[Label+bp]	SS
	[Label+di]	DS
	[Label+si]	DS
	[Label+bx+di]	DS
	[Label+bx+si]	DS
	[Label+bp+di]	SS
[Label+bp+si]	SS	

091.jpg

Alle drei aufgeführten Möglichkeiten der Adressierung im Arbeitsspeicher können zusätzlich noch durch ein festes Displacement ergänzt werden.

Mögliche Kombinationen sind:

```
mov [Tabelle + bx + 6], bl      ; Verarbeitung von 8-Bit-Größen (BL = Byte)
mov [Tabelle + (3 * 2) + bx], bx ; Verarbeitung von 16-Bit-Größen (BX = Wort)
mov [bp + di + 22], ax
mov [Tabelle + 8], 8
```

Wenn mit einem festen Displacement gearbeitet wird, sollte man das Displacement nicht als Wert, sondern über eine Konstante im Befehl angeben. Dies fördert die Übersicht und Wartungsfreundlichkeit des Programms. Eine Konstante wird mit Hilfe des EQU-Befehls im Datenteil des Programms angelegt.

Beispiel

```
; <Wir sind im Daten-Segment>
Tablaeng      equ    100      ; Länge der Tabelle
EndeKZ       equ    '$'      ; Endekennzeichen der Tabelle
; <Wir sind im Code-Segment>
mov  [Tabelle + Tablaeng] ; EndeKZ ; Die beiden Befehle erfüllen den gleichen Zweck
mov  [Tabelle + 100], '$' ; Das Obige ist jedoch wartungsfreundlicher
```

4.7 Übung 4

1. Aufgabe

a) Nennen Sie die vier Allzweckregister des 8086-Prozessors und die Vorteile, die diese gegenüber den anderen Registern haben!

2. Aufgabe

- a) In welchem Segment befinden sich die Maschinenbefehle eines Programms?
b) Welche Register werden für die Adressierung der Maschinenbefehle benötigt?

3. Aufgabe

- a) Welche Größe kann ein Segment maximal einnehmen?
b) Wie wird die 20-Bit-Arbeitsspeicheradresse physikalisch aus dem Segmentregister und dem Offsetregister gebildet?

4. Aufgabe

- a) Was versteht man unter einem Stack?
b) Welche Aufgabe hat das Statusregister?
c) Bei der Registeradressierung wird standardmäßig das Datensegment als Segmentregister verwendet, mit Ausnahme welchen Registers?

4.8 Lösung 4

1.a) Die vier Arbeits- oder auch Allzweckregister des 8086-Prozessors heißen:

AX = Akkumulator

BX = Basisregister

CX = Zählerregister

DX = Datenregister

Alle vier Arbeitsregister lassen sich zusätzlich in zwei 8-Bit-Register unterteilen.

2.a) Die Maschinenbefehle werden in einen Speicherbereich gestellt, der Codesegment heißt.

2.b) Die physikalische 20-Bit-Adresse, um die Maschinenbefehle im Codesegment adressieren zu können, wird aus dem Codesegment (CS) als Basisadresse in Verbindung mit dem Befehlszeiger (IP) gebildet (CS:IP).

3.a) Ein Segment kann 64 Kilobyte einnehmen.

3.b) Physikalische Arbeitsspeicheradresse = Segmentadresse im Adressrechner um vier Bits nach links schieben (Multiplikation mit 16) + Offsetadresse.

4.a) Der Stack ist ein maximal 64 Kilobyte großer Arbeitsspeicherbereich, in dem der Prozessor und der Programmierer kurzzeitig Daten Zwischenspeichern können.

4.b) Das Statusregister zeigt Informationen über das Ergebnis eines Maschinenbefehls an.

4.c) Ist das BP-Register (BP oder Base Pointer) an der Bildung der effektiven Adresse beteiligt, wird als Basisregister das Stacksegment (SS) verwendet (SS:BP).

5 EIN ASSEMBLER-PROGRAMM AUFBAUEN

5.1 Segmente eines Programms

Damit der Computer arbeiten kann, braucht er Daten und Befehle, die ihm sagen, was er tun soll. Ein Assemblerprogramm besteht dazu aus Programmbeehlen und Daten in Form von Variablen. Zusätzlich besitzen die meisten Programme noch einen Zwischenspeicher, den Stack.

Der Stack wird in den Hochsprachen automatisch vom Compiler angelegt und verwaltet. Bei der Programmierung in Assembler müssen wir den Stack selbst anlegen und pflegen. Als Ausgleich dafür haben wir die volle Kontrolle über den Stack und können ihn für eigene Zwecke einsetzen, was der Hochsprachenprogrammierer nicht kann.

Damit alles seine Ordnung hat, werden Daten, Programmbeehle und der Stack in eigene Speicherbereiche getrennt, die Segmente. Der 8086-Prozessor besitzt die vier Segmente DS, ES, CS und SS.

Das Datensegment (DS) und das Extrasegment (ES)

Diese beiden Segmente werden für die Aufnahme der Daten und Variablen herangezogen. Das Extrasegment ist nur von Bedeutung, wenn der Platz im Datensegment zu klein wird.

Das Codesegment (CS) und das Stacksegment (SS)

Das Segment, in dem der Programmcode angesiedelt ist, heißt sinnigerweise Codesegment (CS). Im Codesegment befinden sich die Maschinenbeehle des Programms. Das letzte verfügbare Segment ist das Stacksegment. Im Stacksegment (SS) befindet sich der interne Programmspeicher, der Stack oder Stapel.

Um die Segmente im Programm zu definieren, müssen wir uns des Segmentbeehls *SEGMENT* bedienen, der den Anfang eines Segmentes definiert. Ein Segmentbeehl erzeugt keinen Maschinencode. Er teilt dem Assembler mit, dass ein Speicherbereich angelegt wird, der die im Segment angelegten Daten aufnehmen soll. Ein Segment kann dabei einen beliebigen Namen erhalten. Damit der Assembler weiß, wann ein Segment abgeschlossen ist, geben wir zum Schluss noch einmal den Namen des Segmentes mit dem Segmentbeehl *ENDS* an.

Nach der Definition aller benötigten Segmente in einem Programm, müssen wir den Assembler noch darüber informieren, welches Segmentregister einem Segment zugeteilt werden soll. Dazu gibt es die Segmentanweisung *ASSUME*, die im Codesegment des Programms angegeben werden muss. Die Anweisung *ASSUME* wird in den Maschinencode umgewandelt und teilt dem Assembler beim Start des Programms mit, welches Segmentregister beispielsweise mit dem Datensegment zusammenarbeitet.

Im folgenden Bild ist das Grundgerüst eines Assemblerprogrammes dargestellt.

Grundgerüst eines Assemblerprogrammes

```
DATEN          SEGMENT          ;Anfang des Segments
; <In diesem Segment stehen die Daten und Variablen>
DATEN          ENDS              ; Ende des Segments

STAPEL         SEGMENT
; < In diesem Segment wird der Stack des Programms angelegt >
STAPEL         ENDS

CODE           SEGMENT
; <In diesem Segment stehen die Befehle des Programms >
      ASSUMECS:CODE, DS:DATEN, ES:NOTHING, SS:STAPEL
Anfang:
      mov ax, DATEN
      mov ds, ax                ; Datensegment laden
; < Programmteil >
EndePrg:
      mov ah, 4Ch                ; Ende des Programms
      int 21h
CODE           ENDS
      END      Anfang          ; Start des Programms beim Label Anfang:
```

Betrachten wir das Beispielprogramm etwas näher:

Zuweisung der Segmente an die Segmentregister

In der obigen Abbildung sehen Sie im Codesegment gleich als erste Anweisung den Befehl *ASSUME*. Durch diesen Befehl wird der Assembler darüber informiert, welche Segmentregister für die Segmente zugeteilt sind. Da das Extrasegment in diesem

Programm nicht definiert wurde, ist die Beziehung zum Register ES mit dem Wort *NOTHING* beschrieben. Dies ist eigentlich nicht notwendig, wegen der Übersichtlichkeit des Programms sollten Sie es jedoch angeben.

Datensegment laden

Bevor Sie auf die Daten im Datensegment zugreifen können, müssen Sie das Datensegment erst laden. Da ein Segmentregister technisch nicht mit einem Wert direkt geladen werden kann, muss dieser über ein Register dem Segmentregister zugeführt werden.

Die Befehlsfolge *mov ds, DATEN* funktioniert leider nicht. Sie müssen schon den Umweg über ein zweites Register nehmen. Mit der Befehlsfolge 1. *mov ax, DATEN* 2. *mov d, ax* zum Laden des Segmentregisters ist der Assembler wieder versöhnt. Das Datensegment ist aber das einzige Segment, welches Sie im Programm selber laden müssen. Das Codesegment wird vom Betriebssystem geladen, da sonst der Programmstart vom Prozessor gar nicht gefunden werden könnte. Auch das Stacksegment wird vor dem Programmstart vom Betriebssystem versorgt.

Programmende und zurück zum Betriebssystem

Durch den Aufruf des Programmnamens über die Kommandozeile wird das Maschinenprogramm vom Betriebssystem in den Arbeitsspeicher geladen und dort ausgeführt.

Um ein Programm ordnungsgemäß zu beenden, müssen Sie wiederum das Betriebssystem informieren. Dazu wird der hexadezimale Wert *4Ch* mit dem Befehl *mov ah, 4Ch* in das AH-Register geladen und das Betriebssystem mit dem Befehl *int 21h* aufgerufen. Das Betriebssystem schaut dann im AH-Register nach, welche Routine ausgeführt werden soll. Durch den Übergabeparameter *4Ch* im AH-Register wird dem Betriebssystem mitgeteilt, das Programm zu beenden und selbst wieder die Kontrolle zu übernehmen.

Die Anweisung END

Der Abschluss des Quellprogramms wird mit der Anweisung *END* angezeigt. Alle Zeilen, die hinter der Anweisung *END* stehen, werden vom Assembler ignoriert. Der Anweisung *END* kann zusätzlich ein Label übergeben werden. Dieses Label, im obigen Beispiel mit dem Namen *Anfang* definiert, wird als Startadresse des Programms aufgefasst. Das Programm beginnt also mit dem Befehl *mov ax, DATEN*. Möchten Sie den Programmstart aber an einen anderen Punkt im Programm verlegen, so können Sie dies durch die *END*-Anweisung erreichen. Die Anweisung *END EndePrg* würde in unserem obigen Beispiel den Einstiegspunkt an das Ende des Programms legen. Die ersten und letzten Befehle, die das Programm ausführt, sind dann das Ende des Programms. Folgt der *END*-Anweisung kein Startpunkt, so wird mit dem ersten Befehl im Codesegment begonnen.

5.2 Format einer Assembler-Zeile

Damit der Assembler die Befehle und dazugehörigen Operanden auseinander halten kann, haben die Zeilen im Quelltext folgendes Format:

```
[Label] [Befehl/Anweisung] [Operanden] [;Kommentar]
```

Die eckigen Klammern sollen zeigen, dass die aufgeführten Komponenten einer Assemblerzeile optional sind, d. h. auch weggelassen werden können.

Die Abstände zwischen den einzelnen Komponenten, die als Trennzeichen fungieren, können Sie dabei selbst wählen. Zur Förderung der Übersichtlichkeit eines Programms sollten Sie darauf achten, dass die Spalten, in denen die einzelnen Komponenten definiert werden, einheitlich gestaltet sind

Beispiel

Addiere:

```
add ah, laenge_s
mov cx, 200
```

Für eine gute Übersicht im Programm ist folgende Schreibweise besser geeignet:

Addiere:

```
add ah, laenge_s           ; Lange Satzanfang auf das AH-Register
mov cx, 200                ; Schleifenzähler nach CX
```

Der Aufbau des Quellcodes kann unterschiedlich gestaltet werden.

Wichtig ist vor allem, dass Sie jederzeit einen guten Überblick über Ihr Programm besitzen.

Das Label

Ein Label wird vom Programmierer angelegt, um Variablen, Sprungziele und Werte zu kennzeichnen. Diese Labels werden beim Assemblieren in Adressen umgesetzt.

Im folgenden Beispiel werden die Labels (*Zaehler*, *Max*, *Anfang*, *Schleife* und *Ende*) in den verschiedenen Segmenten definiert:

```

; <Wir sind im Datensegment >
...
Zaehler      dw      ?          ; Variable Zaehler mit 2 Byte anlegen
Max          equ     1000       ; Konstante definieren
...
; <Wir sind im Codesegment >
...
      jmp  Anfang              ; Gehe zu dem Label Anfang:
...
Anfang:
      mov  ax, Zaehler         ; Kopiere Inhalt von Zaehler
Schleife:
      inc  ax                  ; Addiere zum Inhalt AX den Wert 1
      cmp  ax, Max            ; Vergleiche die Inhalte von AX und Max
      je   Ende               ; Wenn AX = Max gehe zu Ende:
      jne  Schleife          ; Wenn nicht, gehe zu Schleife:
Ende:
...

```

Beim Assemblieren werden die Labels durch die entsprechenden Adressen in den Segmenten ersetzt.

```

; <Wir sind im Datensegment >
...
[DS:100]     dw      ?          ; Variable Zaehler mit 2 Byte anlegen
Max          equ     1000       ; Konstante definieren
...
; <Wir sind im Codesegment >
...
      jmp  [CS:200]           ; Gehe zum Label Anfang:
...
[CS:200]:
      mov  ax, [DS:100]      ; Kopiere Inhalt von Zaehler nach AX
[CS:205]:
      inc  ax                  ; Addiere zum Inhalt AX den Wert 1
      cmp  ax, 1000          ; Vergleiche die Inhalte von AX und Max
      je   [CS:225]          ; Wenn AX = Max gehe zu Ende;
      jne  [CS:205]          ; Wenn nicht, gehe zu Schleife:
[CS:225]:

```

Wenn ein Label als Sprungziel verwendet werden soll, muss dieses am Ende einen Doppelpunkt aufweisen. Weiterhin muss dieses Label als erstes Element in der Assemblerzeile aufgeführt werden.

Namensvergabe - die Syntax des Labels

Für ein Label oder für einen Namen in Assembler gelten gewisse Regeln.

Der erlaubte Zeichenvorrat besteht aus:

- den Großbuchstaben A-Z (ohne Umlaute)
- den Kleinbuchstaben a-z (ohne Umlaute)
- den Ziffern 0-9 (nicht am Anfang des Namens)
- den Sonderzeichen _ und @
- den Sonderzeichen ? und \$ (dürfen nicht allein stehen)
- zwischen Groß- und Kleinschreibung wird nicht unterschieden
(alle Labels werden beim Assemblieren in Großbuchstaben umgewandelt)

Zu beachten ist:

- Nur die ersten 31 Zeichen des Labels werden berücksichtigt.
- Ein Label darf nicht durch Leerzeichen unterbrochen werden.

Beispiel

<u>Gültige Namen:</u>	<u>Ungültige Namen:</u>
Zaehler	Zähler
Erster_Anfang	Erster Anfang
Zum_Schluß_?	?
@Stelle_100	100_Stelle

Befehle, Anweisungen und Operanden

Im Mittelpunkt einer Assemblerzeile stehen die Befehle, Anweisungen und die dazugehörigen Operanden. Befehle in der Form *ADD* (addiere) oder *SUB* (subtrahiere) werden vom Assembler in die Maschinsprache übersetzt. Anweisungen, wie beispielsweise die Segmentdefinition oder die Anweisung *END*, werden nicht in Maschinsprache übersetzt, sondern steuern die Arbeitsweise des Assemblers. Dem Befehl oder der Anweisung folgen die benötigten Operanden. Damit der Assembler die Operanden richtig zuordnen kann, müssen diese vom Befehl oder der Anweisung durch mindestens ein Leerzeichen getrennt werden. Werden zwei Operanden angegeben, so müssen diese durch ein Komma voneinander getrennt werden. Pro Zeile wird dabei nur ein Befehl oder eine Anweisung geschrieben, die mit Carriage Return abgeschlossen wird.

Beispiel

<u>Gültiger Befehl:</u>	<u>Ungültiger Befehl:</u>
mov ax, bx	mov ax bx
add ax, Zaehler	addax, Zaehler
sub ax, 5	sub ax, 5 mul bx
mul bx	

Der Kommentar

In jeder Zeile können Sie zusätzlich einen Kommentar angeben. Ein Kommentar fängt dabei mit einem Semikolon an und erstreckt sich über die restliche Zeile. Sie brauchen also kein besonderes Zeichen zu setzen, wenn Sie den Kommentar beenden wollen. Kommentare sollten im Programm so oft wie möglich verwendet werden.

Besonders bei komplizierten Befehlssequenzen ist der Kommentar unerlässlich. Ein guter Assemblerprogrammierer dokumentiert fast jeden Befehl. Denn es ist einfacher und schneller, sich anhand des Kommentars in die Tiefen des Programms vorzuarbeiten, als über die Befehle selbst. Besonders wenn Sie ein altes Programm warten müssen oder wenn Sie ein Programm, das vor längerer Zeit geschrieben wurde, wieder aufnehmen, werden Sie sich über viele Kommentarzeilen freuen. Wenn Sie größere Kommentare für eine Programmbeschreibung mit Ablaufplan in das Programm einbringen wollen, dann können Sie auch die Anweisung *COMMENT* benutzen. Das erste Zeichen nach der Anweisung *COMMENT* wird als Begrenzungszeichen für den Kommentar aufgefasst. Solange dieses Zeichen nicht wieder verwendet wird, können Sie beliebig viele Zeilen Kommentar in den Quellcode einfügen.

Beispiel

```
;Dies ist ein Kommentar, der sich über eine Zeile erstreckt
...
    call Upro          ; Dieser Kommentar beschreibt den Unterprogrammaufruf und den Zweck des
                       ; Unterprogramms
Upro PROC NEAR       ; Anfang des Unterprogramms
...
COMMENT *           Ab hier wird alles als Kommentar aufgefasst
                   Aufgabe:
                   Eingabeparameter:
                   Ausgabeparameter:
                   Info:
    *               ; Ab hier ist wieder alles beim Alten
...
; <Viele Befehle mit Kommentar>
...
    ret              ; Zurück zum Aufrufer
Upro ENDP           ; Ende des Unterprogramms
```

5.3 Variablen und Konstanten anlegen

Für das Anlegen von Variablen in Form von Feldern, Tabellen und Strings steht das Datensegment mit einer Kapazität von 65536 Bytes zur Verfügung. Innerhalb des Datensegmentes können Sie Variablen mit einer Mindestgröße von einem Byte anlegen. Bedingt durch die interne Architektur des Prozessors werden die meisten Variablen in einer Größe von einem Byte (8-Bit-Register), einem Wort (16-Bit-Register) und einem Doppelwort (32-Bit-Register) benötigt. Die BCD-Arithmetik verlangt am häufigsten nach Speichergrößen von 8 und 10 Bytes. Mit Hilfe der nachfolgenden Definitionsanweisungen können Variablen in diesen Größen im Datensegment angelegt werden:

[Variablenname]	[Direktive]	[Ausdruck, ,]
Byte_var	DB	1 ; Define Byte
Wort_var	DW	2 ; Define Word
Doppelwort	DD	4 ; Define Doubleword
Vier_worte	DQ	8 ; Define Quadword
Zehn_bytes	DT	10 ; Define Tenbytes

Durch die Definitionsanweisungen wird im Datensegment der Speicherplatz für die Variable reserviert. Dieser Speicherplatz kann dann über den Variablennamen angesprochen werden.

Zusätzlich sind in unserem Beispiel die Variablen bereits mit einem numerischen Wert initialisiert. Die Variable *Byte_var* beinhaltet den Wert 1, die Variable *Wort_var* den Wert 2 usw. Die Initialisierung der Variablen mit einem Anfangswert ist in vielen Fällen nicht notwendig, da erst im Verlauf des Programms die Variablen mit einem Wert geladen werden. In diesem Fall geben Sie anstelle eines Wertes ein Fragezeichen für den Inhalt der Variablen an. Das Fragezeichen weist den Assembler an, den reservierten Speicherplatz nicht zu initialisieren.

```
Byte_var      DB      ?           ; Reserviere ein Byte im Datensegment
                                   ; ohne Initialisierung dieses Speicherplatzes
```

5.3.1 Zeichenketten

Jedes Programm benötigt Variablen, die Zeichen oder Zeichenketten beinhalten. Da jedes Zeichen aus einem Byte besteht, bietet sich hier die Speicherplatzreservierung mit der Anweisung *define Byte (DB)* an. Damit der Assembler die Vorbelegung als Zeichenkette erkennt, muss diese selbst in einfache oder doppelte Hochkommata gesetzt werden. Möchten Sie in der Zeichenkette selbst einfache oder doppelte Hochkommata verwenden, dürfen die in der Zeichenkette verwendeten Hochkommata nicht als Begrenzungszeichen für den String selbst verwendet werden.

```
Text_1       DB      "Hier geht's"           ; Hier geht's
Text_2       DB      'Auch "so" geht es'     ; Auch "so" geht es
Text_3       DB      "So geht "es" nicht"    ; <-- "Error"
```

Um die Länge des anzulegenden Speicherplatzes brauchen Sie sich nicht selbst kümmern. Der Assembler zählt die in Hochkommata eingeschlossenen Zeichen und reserviert dementsprechend den erforderlichen Speicherplatz.

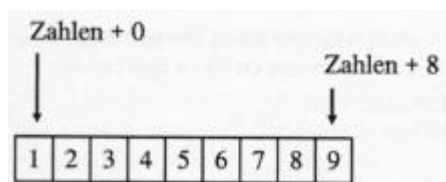
5.3.2 Zahlen

Dasselbe gilt natürlich auch, wenn wir Variablen mit numerischen Werten vorbelegen. Der folgende Befehl zur Speicherplatzreservierung

```
Zahlen       DB      1, 2, 3, 4, 5, 6, 7, 8, 9
```

definiert einen Speicherbereich mit insgesamt neun Bytes. Das erste reservierte Byte enthält den Wert 1, das zweite den Wert 2 usw.

Im Arbeitsspeicher oder genauer im Datensegment schaut die Speicherplatzreservierung mit gleichzeitiger Initialisierung folgendermaßen aus:



104.jpg

Mehrere Bytes werden in diesem Fall unter einem Variablennamen angelegt. Diese Vorgehensweise findet man besonders in der Tabellenverarbeitung. Möchten Sie die einzelnen Felder innerhalb der Tabelle ansprechen, müssen Sie dies dem Assembler über eine Längenangabe mitteilen. Über die direkte Adressierung können Sie dies folgendermaßen realisieren. (Bitte bedenken Sie dabei, dass man in der Assemblerprogrammierung beim Zählen mit 0 anfängt.)

```
; Addiere den Inhalt der Tabelle Zahlen auf das AH-Register
add ah, zahlen           ; AH = AH + zahlen + 0   -> AH = AH + 1
add ah, zahlen + 1      ; AH = AH + zahlen + 1   -> AH = AH + 2
add ah, zahlen + 2      ; AH = AH + zahlen + 2   -> AH = AH + 3
add ah, zahlen + 7      ; AH = AH + zahlen + 7   -> AH = AH + 8
add ah, zahlen + 8      ; AH = AH + zahlen + 8   -> AH = AH + 9
```

Über die indirekte Registeradressierung können wir die Verarbeitung der Tabelle effizienter gestalten:

```
; Addiere den Inhalt der Tabelle Zahlen auf das AH Register
mov cx, 9                ; Die Schleife wird neunmal durchlaufen
mov bx, 0                ; In das BX-Register schreiben wir unser Displacement
schleife:                ; Schleifenanfang
```

```

    add ah, [zahlen + bx]
    inc bx
loop schleife

```

; Erhöhe den Wert im BX-Register um 1 -> BX = BX + 1
; Solange das Register ungleich 0 ist - Springe zu dem Label *schleife*:

5.3.3 DUP-Operator

Stellen Sie sich vor, Sie möchten eine Tabelle mit dem Wert 0 vorbelegen. Bei einer Tabelle mit zehn Elementen dürfte dies ja noch ohne größere Schwierigkeiten zu realisieren sein.

```

Tabelle DB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

```

; Reserviere zehn Bytes und initialisiere jede
; Speicherstelle mit dem Wert 0

Eine Tabelle mit 400 Elementen anzulegen und mit einem Wert zu initialisieren dürfte nicht mehr so schnell von der Hand gehen. Mit Hilfe des DUP-Operators ist aber auch dies kein Problem.

```

Tabelle DB 400 DUP (0)

```

; Reserviere einen Speicherplatz mit 400 Byte und initialisiere
; jedes Byte mit dem Wert 0

Vor dem DUP-Operator muss die Anzahl der Wiederholungen angegeben werden. Der in Klammern stehende Wert initialisiert den reservierten Speicherbereich.

Folgende Beispiele sollen dies verdeutlichen:

		<i>Xmal</i>	<i>wiederhole den Wert in der Klammer</i>					
Tab	DB	10	DUP	(0)	<i>entspricht</i>	Tab	DB	0,0,0,0,0,0,0,0,0,0
Tab	DB	5	DUP	("ft")	<i>entspricht</i>	Tab	DB	"fl","fl","fl","fl","fl"
Tab	DW	10	DUP	(?)	<i>entspricht</i>	Tab	DW	?, ?, ?, ?, ?, ?, ?, ?, ?
Tab	DW	5	DUP	(666)	<i>entspricht</i>	Tab	DW	666, 666, 666, 666, 666
Tab	DB	3	DUP	(1,2,3)	<i>entspricht</i>	Tab	DB	1, 2, 3, 1, 2, 3, 1, 2, 3
Tab	DB	5	DUP	(3*4-2)	<i>entspricht</i>	Tab	DB	10, 10, 10, 10, 10

Um komplexe Wiederholungen zu erzeugen, können DUP-Operatoren auch ineinander verschachtelt werden.

```

Die Anweisung Tab DB 2 DUP (3 DUP (9) ,0 ,2 DUP (1, 2))
entspricht Tab DB 9, 9, 9, 0, 1, 2, 1, 2, 9, 9, 9, 0, 1, 2, 1, 2

```

5.3.4 Konstanten

In den vorherigen Abschnitten dieses Kapitels haben Sie das Definieren und Initialisieren von Variablen kennen gelernt. Wie dort dargelegt, belegen Variablen im Programm Speicherplatz und müssen zur weiteren Verarbeitung vom Datensegment in den Prozessor transferiert werden. Wird innerhalb des Programms mit Direktwerten gearbeitet, so können diese in Form von Konstanten angelegt werden. Der Vorteil einer Konstante ist, dass kein Speicherplatz reserviert werden muss. Nachteilig ist, dass eine Konstante ihren Wert während der Laufzeit des Programms nicht mehr ändern kann. Eine Konstante kann den Wert einer Zahl, eines Zeichens oder einer Zeichenkette repräsentieren. Während der Assemblierung wird die Konstante durch den Wert ersetzt, welchen sie vertritt. Konstanten werden mit Hilfe der EQU-Direktive angelegt.

Die Syntax dieses Befehls lautet:

Name EQU Ausdruck

Dieser Befehl erzeugt keinen Maschinencode und benötigt keinen Speicherplatz. Wichtig dabei ist, dass Sie die Konstante erst definieren, bevor im Programm Bezug auf sie genommen wird. Es empfiehlt sich daher, alle Konstanten an den Anfang des Programms zu stellen.

```

; --- Konstanten-----
Mwst EQU 15 ;Mehrwertsteuer
Int21 EQU int 21h ; Interrupt 21h
Tablen EQU 100 ;Tabellenlange
EndePrg EQU mov ah, 4Ch ; Funktionsnummer für Programmende
Segmzuwei EQU ASSUME CS:CODE, DS:DATEN, SS:STAPEL
OK EQU 0
Fehler EQU 1

; ---Daten-----
DATEN SEGMENT
Tab DB Tablen DUP (0) ; Tabelle mit Tablen Elementen
DATEN ENDS

```

```

; --- Programmcode -----
CODE SEGMENT
    Segmzuwei          ; Segmente zuweisen
Anfang:
    mov ax, DATEN
    mov ds, ax
    mov ah, Mwst       ; Mehrwertsteuer nach Register AH
    mov al, Fehler     ; Fehler -> Returncode nach Register AL
    cmp al, OK         ; Alles OK?
    EndePrg           ; Funktionsnummer für Programmende
    Int 21h           ; Interrupt 21h
CODE ENDS
    END Anfang

```

Sie sollten bei der Arbeit mit Festwerten besonders dann Konstanten einsetzen, wenn sie im Programmcode öfter benötigt werden. Neben der besseren Lesbarkeit Ihres Programms steigern Sie damit auch die Wartungsfreundlichkeit des Sourcecodes. Sollte sich beispielsweise in Ihrem Programm der Mehrwertsteuersatz ändern, so genügt eine Anpassung der Konstante *Mwst*. Ansonsten müssten Sie Ihren gesamten Programmcode nach dem Wert der Mehrwertsteuer durchsuchen. Bei größeren Programmen kann eine derartige Suche leicht zu einem mittleren Nervenzusammenbruch führen. Weiterhin sollte man in jedem Programm immer mit den gleichen Konstanten arbeiten. Hier bietet sich das Sammeln der verschiedenen Konstanten in einer Datei an. Diese angelegte Datei kann über die noch zu besprechende Anweisung *INCLUDE* in jedes Programm kopiert werden, so dass Sie immer mit einem einheitlichen Stand arbeiten.

Name = numerischer Ausdruck

Konstanten können nicht nur durch die EQU-Anweisung angelegt werden. Die zweite Möglichkeit, eine Konstante zu definieren, steht Ihnen mit dem =-Operator zur Verfügung. Mit dem =-Operator können Sie einer Konstante aber nur einen numerischen Wert zuordnen.

Der Vorteil ist, dass Sie den Inhalt der Konstanten beliebig oft verändern können.

Beispiel

```

; --- Konstanten----- ;
Tablen = 100          ;Tabellenlänge

; --- Daten----- ;
DATEN SEGMENT
Tab_1 DB    Tablen DUP    (0)    ; Tabelle mit "Tablen" Elementen
Tablen =    Tablen / 2      ; "Tablen" halbieren
Tab_2 DB    Tablen DUP    (0)    ; Tab_2 mit der Größe Tab_1 / 2
DATEN ENDS

```

5.3.5 INCLUDE-Anweisung

Mit der Hilfe der Anweisung *INCLUDE* können Sie eine Datei, deren Name hinter dem Befehl angegeben werden muss, in den Programmsource kopieren. Der Assembler betrachtet dann diese Datei als Bestandteil des Programms.

In dem oben aufgeführten Beispiel würde es sich anbieten, alle Konstanten in einer separaten Datei anzulegen und sie je nach Bedarf in das jeweilige Listing einzubinden.

Für Include-Dateien sollten Sie der besseren Übersicht wegen die Extension *INC* wählen. Wenn Sie nach der *INCLUDE*-Anweisung nur den Dateinamen angeben, sucht der Assembler die entsprechende Datei im aktuellen Directory. Möchten Sie die Include-Dateien auslagern, so müssen Sie noch zusätzlich einen Pfad angeben.

Die Syntax der Include-Anweisung lautet:

```
INCLUDE [Laufwerk] [\Pfad] [\Dateiname]
```

Beispiel

Datei *Konstant.inc*:

```

Mwst      EQU    14          ; Mehrwertsteuer
Int21     EQU    int 21h     ; Interrupt 21h
Tablen    EQU    100        ; Tabellenlänge
EndePrg   EQU    mov ah, 4Ch ; Funktionsnummer für Programmende
;
Segmzuwei EQU    ASSUME CS:CODE, DS:DATEN, SS:STAPEL
OK        EQU    0

```

```
Fehler EQU 1
```

Datei *Datentab.inc* im Directory *myinc* auf dem Laufwerk *d:*

```
Tab DB Tabellen DUP (0) ; Tabelle mit Tablen Elementen
```

Programmsource:

```
; --- Konstanten -----
INCLUDE konstant.inc

;--- Daten -----
DATEN SEGMENT
INCLUDE d:\myinc\datentab.inc
DATEN ENDS

;--- Programmcode -----
CODE SEGMENT
Segmzuwei ;Segmente zuweisen
Anfang:
    mov ax, DATEN
    mov ds, ax
    ;
    mov ah, Mwst ; Mehrwertsteuer nach Register AH
    ;
    mov bl, Fehler ; Fehler -> Returncode nach Register BL
    ;
    cmp bl, OK ; Alles OK?
    ;
    EndePrg ; Funktionsnummer für Programmende
    ;
    Int 21 ; Interrupt 21h
CODE ENDS
END Anfang
```

5.3.6 \$-Operator

Der \$-Operator steht für die aktuelle Adresse der Programmstelle.

Dieser Wert beinhaltet den Abstand vom Anfang des Segmentes bis zur aktuellen Position, an der der \$-Operator definiert wurde. Dies entspricht dem Offset des nächsten zu assemblierenden Befehls (Anweisung). Das Symbol \$ erweist sich besonders nützlich bei Längenberechnungen im Codesegment.

Im nachfolgenden Beispiel wollen wir den \$-Operator für die Längenberechnung einer Tabelle verwenden. Die Länge soll der Variablen *Tab_1* zugewiesen werden.

```
Tabelle db 1, 2, 3, 4, 5, 6, 7, 8, 9
Tab_1 db $ - Tabelle ; Länge der Tabelle = 9
```

Natürlich können Sie die Länge der Tabelle auch selbst bestimmen. Bei längeren Tabellen kann dies aber recht mühselig sein. Der Hauptvorteil ist, dass Sie jeder Änderung, die die Tabelle betrifft, sorglos entgegensehen können. Eine Korrektur in der Tabelle hat auch die automatische Anpassung der Länge der Tabelle zur Folge.

5.3.7 ORG-Anweisung

Durch die ORG-Anweisung haben Sie die Möglichkeit, während der Assemblierung die aktuelle Offset-Adresse zu steuern. Die Syntax dieses Befehls lautet:

ORG Adresse

Als Adresse können Sie einen Direktwert (Ganzzahl) oder ein Label angeben. Wird ein Direktwert angegeben, muss dieser zwischen 0 und 65535 liegen. Wird ein Label in Form einer Variablen oder einer Sprungadresse angegeben, muss das Label vorher definiert worden sein. Die ORG-Anweisung eignet sich besonders gut zur Redefinition von Speicherstellen.

; <Wir befinden uns im Daten-Segment>

```
Datum db "26.01.1993" ; Reserviere 10 Byte Speicherplatz
org Datum ; Gehe zu der Adresse des Labels Datum
```

```

Tag1  db    ?           ; Tag1 zeigt jetzt auf die Adresse, die als Inhalt eine 2 hat
Tag2  db    ?           ; Tag2 zeigt auf die Adresse, die als Inhalt eine 6 hat
org   $ + 8           ; Erhöhe die aktuelle Offsetadresse um 8
                        ; und überspringe mit Hilfe des $-Operators dadurch den Teil .01.1993

```

; <Wir befinden uns im Code-Segment>

```

...
mov  Tag1, "1"
mov  Tag2, "2"           ; Neues Datum ist der 12.01.1993
...

```

5.4 Übung 5

Aufgabe

Im nachfolgenden Programm haben sich drei Fehler eingeschlichen.
Suchen Sie diese Fehler!

```

DATEN SEGMENT
Var1  db    1200        ; Definiere Var1 und Var2 und
Var2  db    255         ; belege diese mit einem Wert vor
Var3  dw    ?          ; Definiere Var3 ohne Initialisierung
DATEN ENDS

```

.***** Stack *****

```

STAPEL SEGMENT
      dw    128         dup    (?)
STAPEL ENDS

```

.***** Code HP *****

```

CODE SEGMENT
      ASSUME CS:CODE, DS:DATEN, SS:STAPEL
Anfang: mov  ax, DATEN
        mov  ds, ax      ; Datensegment zuordnen
        mov  ax, Var1    ; Schiebe Inhalt von Var1 nach AX
        mov  bx, Var2    ; Schiebe Inhalt von Var2 nach BX
        add  ax, bx      ; AX = AX + BX
        mov  Var3, ax    ; Schiebe Inhalt von AX nach var3
        int  21h        Funktion aufrufen
CODE ENDS
      END Anfang

```

5.5 Lösung 5

1. Fehler:

Die Variable Var1 hat eine Größe von einem Byte.
Der maximale Wert, den ein Byte aufnehmen kann, liegt bei einer Größe von 255.

2. Fehler:

Bei dem Befehl `mov bx Var2` fehlt das Komma als Begrenzer der Operanden.
Richtig wäre der Befehl `mov bx,Var2`.

3. Fehler:

Der Kommentar `Funktion aufrufen` muss am Anfang mit einem Semikolon gekennzeichnet werden.

6 WICHTIGE BEFEHLE

Sie haben sich mit den Grundlagen der Assembler-Programmierung vertraut gemacht. Im folgenden Kapitel wollen wir die grundlegenden Assembler-Befehle durchgehen und uns dabei neben der Theorie mit der Praxis auseinandersetzen. Es sollen diejenigen Befehle behandelt werden, die in der Praxis am häufigsten benötigt werden.

Eine alphabetisch sortierte Befehlsübersicht des 8086-Prozessors finden Sie im Anhang dieses Buches.

Alle Befehle werden mit den benötigten Operanden aufgeführt. Die meisten Befehle verändern das Statusregister des Prozessors. Die durch den Befehl veränderten Statusflags werden neben dem Befehl in Klammern aufgeführt.

6.1 Verschiebe- und Ladebefehle

Diese Befehle werden in der Assembler-Programmierung sehr häufig eingesetzt. Wie die Namen schon verraten, werden durch diese Anweisungen Daten vom Quelloperanden zum Zieloperanden verschoben (kopiert). Wahlweise kann es sich beim Quell- oder Zieloperanden um eine Speicherstelle oder ein CPU-Register handeln

6.1.1 Transportbefehl MOV

MOV Ziel, Quelle (< --)

Der Datentransferbefehl *MOV* ist vom englischen Wort *move* abgeleitet und überträgt ein Byte oder Wort vom Quelloperanden zum Zieloperanden. Das Statusregister wird durch den MOV-Befehl nicht verändert.

Zu den am häufigsten eingesetzten Befehlen gehört der MOV-Befehl. Der Befehl MOV kopiert den im Quelloperanden angegebenen Wert in den Zieloperanden, ohne dass der Wert im Quelloperanden verändert wird. Wie Sie aus der Abbildung ersehen können, wird beim MOV-Befehl wie bei fast allen Zwei-Operanden-Befehlen des 8086 zuerst das Ziel des Datentransportes angegeben. Das Ziel ist dabei ein CPU-Register oder eine Speicherstelle. Als Quellangabe können Sie neben einem CPU-Register oder einer Speicherstelle auch einen Direktwert angeben. Quell- und Zieloperand müssen dabei die gleiche Größe aufweisen. Der Befehl `mov ah, bx` würde nicht funktionieren, da die Übertragung einer 16-Bit-Größe auf eine 8-Bit-Größe syntaktisch und logisch falsch ist. Weiterhin kann auf die einzelnen Flagregister und die Register CS und IP mit dem MOV-Befehl nicht zugegriffen werden. Auch ist nicht jede beliebige Kombination der Ziel- und Quelloperanden möglich. Das direkte Verschieben des Inhalts einer Speicherzelle in eine andere Speicherzelle ist nicht ohne Umweg über ein CPU-Register realisierbar. Folgende Abbildung zeigt die möglichen Operandenkombinationen des MOV-Befehls:

MOV Ziel,	Quelle	
CPU-Register	CPU-Register	; mov ax, bx
CPU-Register	Segment-Register	; mov cx, ds
CPU-Register	Speicher	; mov cx, zaehler
CPU-Register	Wert	; mov ah, 255
Segment-Register	CPU-Register	; mov es, ax
Segment-Register	Speicher	; mov ss, save, ss
Speicher	CPU-Register	; mov zaehler, cx
Speicher	Segment-Register	; mov save_es, es
Speicher	Wert	; mov zaehler, 0

Die nachfolgende Abbildung zeigt die äquivalenten Hochsprachenbefehle zum Assembler-Befehl MOV, der in diesem Beispiel die Variable *Zaehler* mit dem Wert 0 belegt.

<code>mov zaehler, 0</code>	; Assembler	<code>zaehler = 0</code>	/*C*/
<code>zaehler := 0</code>	{ Pascal }	*	Cobol
		<code>MOVE ZERO TO zaehler</code>	

6.1.2 Tauschbefehl XCHG

XCHG Operand1, Operand2 (< -- >)

Abgeleitet vom Wort *exchange* tauscht der XCHG-Befehl die Inhalte von Operand1 mit dem Inhalt von Operand2. Das Statusregister wird durch den XCHG-Befehl nicht verändert.

Häufig müssen in Sortier Routinen der Inhalt von Registern und Speicherstellen gegeneinander ausgetauscht werden. Der XCHG-Befehl bewirkt, dass der Inhalt von Operand1 in einem internen Register zwischengespeichert wird. Danach wird der Inhalt von Operand2 nach Operand1 übertragen. Zum Abschluss wird der im internen Register zwischengespeicherte Inhalt von Operand1

nach Operand2 übertragen. Die beiden Operanden müssen dabei die gleiche Größe aufweisen. Beim XCHG-Befehl dürfen Sie keine Segmentregister verwenden. Der Befehl *XCHG Speicher, Speicher* ist nicht zulässig.

Folgende Abbildung zeigt die möglichen Operandenkombinationen des XCHG-Befehls:

XCHG	Operand1,	Operand2	
	CPU-Register	CPU-Register	;xchg ax, bx
	CPU-Register	Speicher	;xchg ex, Zaehler
	Speicher	CPU-Register	;xchg Zaehler, ex

Beispiel

```
xchg al, ah ; Vertausche die Inhalte der Register AL und AH (AL<-->AH)
```

ist mit der folgenden Befehlsfolge gleichzusetzen:

```
mov bh, al ; Inhalt von Register AL Zwischenspeichern
mov al, ah ; Inhalt des Registers AH nach AL
mov ah, bh ; Inhalt des Registers BH nach AH
```

6.1.3 Ladebefehl LEA

LEA Ziel, Quelle (*<-- Offsetadresse des Quelloperanden*)

Der LEA-Befehl (engl. load effective adress) lädt die Offsetadresse des Quelloperanden in den Zieloperanden. Das Statusregister wird durch den LEA-Befehl nicht verändert.

Der LEA-Befehl überträgt die Offsetadresse (effektive Adresse) in ein 16-Bit-Register, welches im Zieloperanden angegeben wird. Als Quelloperand wird meist ein Label verwendet.

Häufig benutzt man den LEA-Befehl in Verbindung mit der indirekten Register-Adressierung, um Tabellen oder Strings zu verarbeiten. Der LEA-Befehl kann durch den MOV-Befehl in Verbindung mit dem Offset-Operator ersetzt werden. Nachfolgende Befehle erfüllen den gleichen Zweck:

```
; Lade die Speicheradresse, an der die Variable Tabelle definiert ist.
; In diesem Beispiel soll die Variable Tabelle an der Speicherstelle 200 stehen.
```

```
Tabelle DB 100 DUP (0) ; Variable mit 100 Bytes definieren
;
...
lea bx, Tabelle ; Lade den Offset der Variable Tabelle ist gleichbedeutend zu
mov bx, offset Tabelle ; In diesem Fall 200 nach
; Register BX
...
```

Bei der Programmierung sollten Sie den LEA-Befehl bevorzugen, da man häufig den Offset-Operator beim MOV-Befehl übersieht.

Der LEA-Befehl kann nicht nur für die indirekte Adressierung in Verbindung mit Variablen aus dem Datenssegment benutzt werden.

```
lea dx, Ende_Prg ;Lade die effektive Adresse des Labels
;Ende_Prg: in das DX-Register
jmp dx ; Springe zu der Adresse die im DX-Register steht 1.)

Ende_Prg: ; <----- I.)
mov ah, 4Ch
int 21h
```

Wie Sie sehen, ist es für den LEA-Befehl unerheblich, in welchem Segment das Label vereinbart wurde. Der LEA-Befehl berechnet immer die effektive Adresse im Segment, in dem das Label definiert wurde.

6.1.4 Stack-Befehle PUSH und POP

Die Stack-Befehle PUSH und POP ermöglichen den Datentransport eines Wort-Operanden in Form eines Registers oder einer Variablen auf den Stack (siehe Kapitel 4.4).

Am häufigsten werden die Stack-Befehle für das kurzzeitige Zwischenspeichern von Registerinhalten benutzt, da die Anzahl der Register selten ausreicht, um alle Aufgaben damit zu bewältigen. Um den Inhalt eines Registers oder einer Variablen auf den

Stack zu transportieren, wird der PUSH-Befehl benötigt. Die Syntax dieses Befehls lautet:

PUSH Quelle ;Inhalt Quelle --> Stack

Durch den PUSH-Befehl wird der Inhalt des angegebenen Quelloperanden an die Spitze des Stacks transportiert. Der Stack-Pointer wird dazu um den Wert zwei gesenkt und zeigt damit auf eine freie, 2 Byte große Speicherstelle. Anschließend wird der Inhalt des Quelloperanden in die freie Speicherstelle eingetragen.

Um den Inhalt wieder vom Stack zu nehmen, steht der POP-Befehl zur Verfügung. Die Syntax dieses Befehls lautet:

POP Ziel ;Inhalt Stack --> Ziel

Der POP-Befehl überträgt zwei Bytes in den angegebenen Zielperanden (Register/Speicher). Danach wird der Stack-Pointer um den Wert zwei erhöht und zeigt damit auf den nächsten Eintrag des Stacks.

Beachten Sie bitte, dass der Stack nach dem LIFO-Prinzip (vgl. Kapitel 4.4) arbeitet. Die zwischengespeicherten Operanden müssen Sie daher beim Zurückholen in umgekehrter Reihenfolge vom Stack nehmen. Nachfolgendes Beispiel zeigt diese Vorgehensweise:

```
1.) --> push ax      ; Ersten Inhalt des AX-Registers auf den Stack kopieren,
2.) --> push bx      ; ebenso den Inhalt des BX-Registers (2.)
...
...                ; <Befehle>
...
2.) --> pop bx       ; Inhalt (2.) des Stacks in das BX-Register kopieren
1.) --> pop ax       ; und den nächsten Wert (1.) vom Stack holen
```

In dieser Reihenfolge werden die Anfangswerte der Register wiederhergestellt.

6.2 Mathematische Grundbefehle

In diesem Kapitel wollen wir uns mit der Hauptaufgabe eines Computers beschäftigen. Der Name Computer kommt vom englischen Ausdruck to compute, was soviel bedeutet wie rechnen. Zwar ist der Intel-Prozessor 8086 nicht gerade ein Rechengenie, er beherrscht nur die Grundrechenarten Addition, Subtraktion, Multiplikation und Division. Das ist jedoch ausreichend, denn alle höheren mathematischen Operationen, wie beispielsweise Potenzfunktionen oder Wurzelfunktionen, lassen sich aus diesen Grundrechenarten ableiten.

6.2.1 Additionsbefehl ADD

ADD Ziel,Quelle ;Ziel = Ziel + Quelle (OF,SF,ZF,AF,PF,CF)

Der ADD-Befehl addiert den Inhalt des Zielperanden mit dem Inhalt des Quelloperanden und legt das Ergebnis im Zielperanden ab.

Der ADD-Befehl arbeitet dabei mit 8- oder 16-Bit-Operanden. Beide Operanden müssen die gleiche Größe aufweisen. Wird der Wertebereich des Zielperanden durch die Addition überschritten, wird dies durch das Statusregister angezeigt.

Besonders wichtig ist wegen des möglichen Überlaufs des Wertebereichs hier die Berücksichtigung des Carry-Flags.

Folgende Abbildung zeigt die möglichen Operandenkombinationen des ADD-Befehls:

ADD	Ziel,	Quelle	
	CPU-Register	CPU-Register	; add ax, bx
	CPU-Register	Speicher	; add ax, Zaehler
	CPU-Register	Wert	; add ah, 25
	Speicher	Register	; add Zaehler, ax
	Speicher	Wert	; add Zaehler, 11

Beispiel

```
add ax, 2000    ; AX = AX + 2000
add Zaehler, 10 ; Zaehler = Zaehler + 10
add al, 11      ; AL = AL + 11
```

6.2.2 Subtraktionsbefehl SUB

SUB Ziel,Quelle ; ;Ziel = Ziel - Quelle (OF,SF,ZF,AF,PF,CF)

Der SUB-Befehl subtrahiert den Inhalt des Quelloperanden vom Inhalt des Zielperanden und legt das Ergebnis im Zielperanden ab. Der SUB-Befehl arbeitet dabei mit 8- oder 16-Bit-Operanden. Beide Operanden müssen die gleiche Größe aufweisen.

Folgende Abbildung zeigt die möglichen Operandenkombinationen des SUB-Befehls:

SUB	Ziel,	Quelle	
	CPU-Register	CPU-Register	; sub ax, bx
	CPU-Register	Speicher	; sub ax, Zaehler
	CPU-Register	Wert	; sub ah, 25
	Speicher	Register	; sub Zaehler, ax
	Speicher	Wert	; sub Zaehler, 10

Beispiel

```
sub ax, 2000 ; AX = AX - 2000
sub Zaehler, 10 ; Zaehler = Zaehler - 10
sub al, 11 ; AL = AL - 11
```

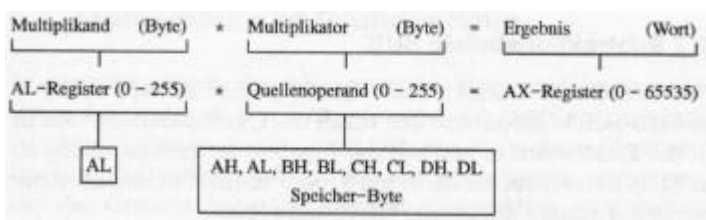
6.2.3 Multiplikationsbefehl MUL

MUL Quelle (OF,SF=?,ZF=?,AF=?,PF=?,CF)

Der Befehl MUL multipliziert den Wert im Quelloperanden mit dem Wert im Akkumulator. Als Quelloperand kann ein CPU-Register oder eine Variable verwendet werden. Beim MUL-Befehl müssen wir zwischen einer Byte- und einer Wort-Multiplikation unterscheiden.

Die Byte-Multiplikation multipliziert zwei vorzeichenlose 8-Bit-Werte miteinander und legt das Ergebnis im AX-Register ab. Ein Operand muss sich dabei im AL-Register befinden. Der zweite Operand, der aus einem 8-Bit-Register oder einer Byte-Speicherstelle bestehen muss, wird beim Aufruf des MUL-Befehls mit angegeben.

Multiplikand(Byte) * Multiplikator(Byte)



120.jpg

Beispiele

In dem folgenden Beispiel wird das BH-Register mit dem AL-Register multipliziert und das Ergebnis (5000) im AX-Register abgelegt:

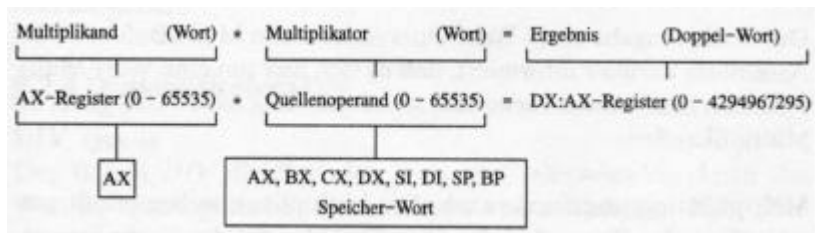
```
Ergebnis      dw      ?      ; Variable Ergebnis anlegen (2 Byte)
...
mov  al, 50      ; Multiplikand ist immer das Register AL
mov  bh, 100     ; Wert 100 als Multiplikator nach Register BH
mul  bh          ; Wert im Register AL * Wert im Register BH
mov  Ergebnis, ax ; Ergebnis der Multiplikation steht im Register AX
```

Durch die Angabe eines Byte-Operanden beim MUL-Befehl weiß der Assembler, dass es sich hier um eine Byte-Multiplikation handelt, und verwendet standardmäßig das AL-Register als Multiplikanden. Im nächsten Beispiel wollen wir das AL-Register mit dem Wert 80 quadrieren:

```
Ergebnis      dw      ?      ; Variable Ergebnis anlegen (2 Byte)
...
mov  al, 80      ; AL = 80
mul  al          ; AL * AL = AX
mov  Ergebnis, ax ; Ergebnis = AX
```

Die Wort-Multiplikation multipliziert zwei vorzeichenlose 16-Bit-Werte miteinander und legt das Ergebnis im Registerpaar DX:AX ab. Der höherwertige Wert steht dabei im DX-Register, der niederwertige Wert im AX-Register. Ein Operand muss sich im AX-Register befinden. Der zweite Operand, der aus einem 16-Bit-Register oder einer Wort-Speicherstelle bestehen muss, wird beim Aufruf des MUL-Befehls mit angegeben.

Beispiel



121.jpg

In diesem Beispiel wird das AX-Register mit einer Wort-Speicherstelle multipliziert und das Ergebnis im Registerpaar DX:AX abgelegt. Das Ergebnis besteht aus einer 32-Bit-Größe. Da ein Transport von 32-Bit-Werten erst ab dem 386-Prozessor möglich ist, müssen wir auf die indizierte Adressierung zurückgreifen, um das Ergebnis im Registerpaar DX:AX in einer Doppel-Wort-Speicherstelle (4 Byte) zu sichern.

Vorsicht: Nach der Intel-Konvention wird ein Doppel-Wort in Form zweier Worte abgespeichert. Die Bits 0 bis 15 des Doppel-Wortes werden als niederwertiges Wort der 32-Bit-Zahl im Speicher vor dem höherwertigen Wort abgelegt. Im Speicher wird ein Doppel-Wort mit Niederwertiges Wort - Höherwertiges Wort abgelegt. Dies muss beim Zugriff auf 32-Bit-Werte unbedingt beachtet werden.

```

Szahl          dd      ?      ; Variable Szahl anlegen (4 Byte)
Wort_var       dw      ?      ; Variable Wort_var anlegen (2 Byte)
...
mov  ax, 3000          ; AX = 3000
mov  Wort_var, 300     ; Wort_var = 300
mul  Wort_var          ; AX (3000) * Wort_var (300) = DX:AX (900.000)
mov  word ptr Szahl + 2, dx ; Höherwertigen Teil des Ergebnisses in die letzten zwei Bytes der Variable
                                ; Szahl transportieren
mov  word ptr Szahl, ax ; Niederwertigen Teil des Ergebnisses in die ersten zwei Bytes der Variable
                                ; Szahl transportieren

```

Durch die Angabe eines Wort-Operanden beim MUL-Befehl ist der Assembler darüber informiert, dass es sich hier um eine Wort-Multiplikation handelt und verwendet standardmäßig das AX-Register als Multiplikanden.

Multiplikation und Steuerwerk: Die Multiplikation beeinflusst ausschließlich das Carry- und das Overflow Flag. Ist das Carry Flag gesetzt, bedeutet dies, dass ein Überlauf eingetreten ist und das Ergebnis nicht mehr richtig dargestellt werden kann. Bei einer Byte-Multiplikation würde das Carry Flag anzeigen, ob das Ergebnis größer 255 bzw. das Ergebnis einer Wort-Multiplikation größer 65535 ist. Byte-Multiplikationen haben als Ergebnis immer einen 16-Bit-Wert im AX-Register, Wort-Multiplikationen als Ergebnis immer einen 32-Bit-Wert im Registerpaar DX:AX. Das Carry Flag zeigt somit zwar richtig an, dass der ursprüngliche Wertebereich überschritten wurde. Das Ergebnis der Multiplikation ist aber richtig. Mit dem Carry-Flag kann bei der Multiplikation also nur überprüft werden, ob der ursprüngliche Wertebereich überschritten wurde.

Beispiel

```

Ergebnis     dw      ?      ; Variable Ergebnis (2 Byte)
...
mov  al, 30      ; AL = 30
mov  bh, 10     ; BH = 10
mul  bh          ; AX = AL * BH
je  ueberlauf   ; ist AX > 0 - 255 - Jump on carry (jc) zu dem Label ueberlauf:
...
ueberlauf:
...
mov  Ergebnis, ax ; Ergebnis = Inhalt des AX-Registers

```

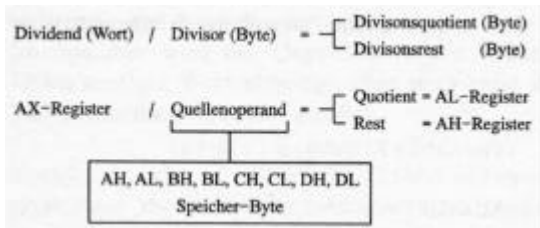
Durch die obige Byte-Multiplikation wird der Wertebereich des AL-Registers überschritten, was das Carry Flag auch richtig anzeigt. Das Ergebnis der Multiplikation wird aber im AX-Register richtig dargestellt. Das gleiche gilt für die Wort-Multiplikation. Hier zeigt das Carry Flag an, dass der Wertebereich des AX-Registers überschritten worden ist. Das Ergebnis im Registerpaar DX:AX ist korrekt dargestellt.

6.2.4 Divisionsbefehl DIV

DIV Quelle

Der Befehl *DIV* dividiert den Wert im Zieloperanden durch den Wert im Quelloperanden. Der Zieloperand ist bei einer Byte-Division standardmäßig das AX-Register, bei einer Wort-Division das Registerpaar DX:AX. Als Quelloperand kann ein CPU-Register oder ein Speicheroperand verwendet werden.

Die Byte-Division dividiert den Inhalt des AX-Registers durch einen 8-Bit-Wert. Das Ergebnis der Operation wird im AX-Register abgelegt. Der Quotient, also das ganzzahlige Ergebnis der Division, befindet sich im AL-Register, der Divisionsrest im AH-Register. Der Divisor, der aus einem 8-Bit-Register oder einer Byte-Speicherstelle besteht, muss beim Aufruf des DIV-Befehls angegeben werden.



124.jpg

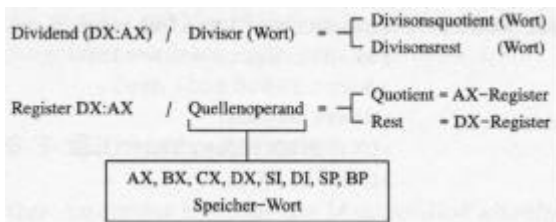
Beispiel

Wir wollen den 16-Bit-Wert 4522 im AX-Register durch den 8-Bit-Wert 120 dividieren. Das Ergebnis 37,82 wird dabei auf die beiden 8-Bit-Register im Akkumulator aufgeteilt. Das AL-Register enthält das ganzzahlige Divisionsergebnis 37, das AH-Register den Divisionsrest 82.

```
mov ax, 4522      ; Dividend ist immer das AX-Register
mov bh, 120      ; durch den Wert 120 soll geteilt werden
div bh           ; AX / BH = Quotient AL - Rest in AH
```

Durch die Angabe eines Byte-Operanden beim DIV-Befehl weiß der Assembler, dass es sich hier um eine Byte-Division handelt, und verwendet standardmäßig das AX-Register als Dividenten.

Die Wort-Division dividiert den Inhalt des Registerpaares DX:AX durch einen 16-Bit-Wert. Das Ergebnis der Operation wird im Registerpaar DX:AX abgelegt. Der Quotient, also das ganzzahlige Ergebnis der Division, befindet sich im AX-Register, der Divisionsrest im DX-Register. Der Divisor, der aus einem 16-Bit-Register oder einer Wort-Speicherstelle besteht, muss beim Aufruf des DIV-Befehls angegeben werden.



125.jpg

Beispiel

Wir wollen den 16-Bit-Wert 45220 im AX-Register durch den 16-Bit-Wert 120 dividieren. Das Ergebnis 376,100 wird dabei auf die beiden 16-Bit-Register aufgeteilt. Das AX-Register enthält das ganzzahlige Divisionsergebnis 376, das DX-Register den Divisionsrest 100.

```
mov dx, 0        ; DIV Wort nimmt immer DX:AX als Dividenten
mov ax, 45220    ; eigentlicher Divident
mov bx, 120      ; durch den Wert 120 soll dividiert werden
div bx          ; DX:AX / BX = Quotient in AX - Rest in DX
```

Als nächstes wollen wir den 32-Bit-Wert 100000 durch den 16-Bit-Wert 35000 dividieren. Das Ergebnis 2,30000 wird dabei auf das Registerpaar DX:AX aufgeteilt. Das AX-Register enthält das ganzzahlige Divisionsergebnis 2, das DX-Register den Divisionsrest 30000.

Vorsicht: Nach der Intel-Konvention wird ein Doppel-Wort in Form zweier Worte abgespeichert. Die Bits 0 bis 15 des Doppel-Wortes werden als niederwertiges Wort der 32-Bit-Zahl im Speicher vor dem höherwertigen Wort abgelegt.

Im Speicher wird ein Doppel-Wort mit Niederwertiges Wort - Höherwertiges Wort abgelegt. Dies muss beim Zugriff auf 32-Bit-Werte unbedingt beachtet werden.

```
Szahl  dd      180000          ; Szahl (4 Byte) mit 180000 vorbelegt
...
mov  dx, word ptr Szahl + 2 ; höherwertiger Teil des 32-Bit-Wertes
mov  ax, word ptr Szahl    ; niederwertiger Teil des 32-Bit-Wertes
mov  bx, 35000             ; durch 35000 soll dividiert werden
div  bx                    ; DX:AX/BX = Quotient in AX - Rest in DX
```

Durch die Angabe eines Wort-Operanden beim DIV-Befehl weiß der Assembler, dass es sich hier um eine Wort-Division handelt. Er verwendet standardmäßig das Registerpaar DX:AX als Dividenten.

6.2.5 Inkrementieren und Dekrementieren

Bei der Tabellenverarbeitung oder in einer Schleife muss häufig ein Zähler um den Wert eins erhöht oder gesenkt werden. Dies erkannten auch die Intel-Entwickler und stellten dafür zwei spezielle Maschinenbefehle zur Verfügung.

INC Ziel ; Ziel = Ziel + 1 (OF,SF,ZF,AF,PF)

Der INCRement-Befehl addiert den Wert 1 zum Ziel. Als Ziel kann ein CPU-Register oder ein Speicheroperand angegeben werden.

Beispiel

```
inc  ax          ; AX = AX + 1
inc  bh          ; BH = BH + 1
inc  zaehler     ; Zaehler = Zaehler + 1
```

DEC Ziel ; Ziel = Ziel - 1 (OF,SF,ZF,AF,PF)

Der DECrement-Befehl subtrahiert den Wert 1 vom Ziel. Als Ziel kann ein CPU-Register oder ein Speicheroperand angegeben werden.

Beispiel

```
dec  ax          ; AX = AX - 1
dec  bh          ; BH = BH - 1
dec  zaehler     ; Zaehler = Zaehler - 1
```

Anmerkung: Die beiden Befehle INC und DEC können auch mit den Befehlen ADD und SUB realisiert werden. Wenn jedoch ein Operand um den Wert 1 erhöht oder gesenkt werden soll, ist es ratsam, die Befehle INC und DEC zu benutzen, da sie erheblich schneller ausgeführt werden.

6.3 Bitmanipulationen

Der Assembler stellt für die Manipulation einzelner Bits in einem Byte oder Wort verschiedene Befehle zur Verfügung. Mit Hilfe dieser Befehle können einzelne Bits gesetzt, gelöscht oder getestet werden. Anwendung finden diese Befehle im besonderen bei dem direkten Zugriff auf die Hardware.

Für das bessere Verständnis dieser Befehle sollten Sie sich die CPU-Register und Speicherzellen nicht als Byte- und Wort-Größen vorstellen, sondern als eine Ansammlung von 8 Bit bei einem Byte und 16 Bit bei einem Wort.

6.3.1 Logische Operationen

Für das Ändern einzelner Bits innerhalb eines Bytes bzw. eines Wortes stellt der Assembler die Befehle AND, OR, XOR und NOT zur Verfügung. Wie diese logischen Befehle die einzelnen Bits manipulieren, können Sie der nachfolgenden Tabelle entnehmen.

Die Arbeitsweise der Befehle AND-OR-XOR-NOT (boolesche Algebra):

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

A	NOT A
0	1
1	0

127.jpg

AND Ziel,Quelle (OF=0,SF,ZF,PF,CF=0)

Der AND-Befehl führt eine logische UND-Verknüpfung mit den Inhalten des Ziel- und Quelloperanden durch. Das Ergebnis der UND-Verknüpfung wird im Zieloperanden abgelegt. Der Zieloperand ist ein CPU-Register oder eine Speicherstelle. Der Quelloperand kann ein CPU-Register, eine Speicherstelle oder ein Direktwert sein. Ziel und Quelle müssen die gleiche Größe aufweisen, und nur ein Operand darf für eine Speicherstelle benutzt werden.

Beispiel

```
and ah, 1111110b ; Setze niederwertiges Bit im AH- Register auf 0
and ax, bx       ; AX = AX AND BX
and dh, 0        ; DH = 0
and Maske, F0h   ; Bits 3-0 in der Variable Maske auf 0 setzen
```

OR Ziel, Quelle (OF=0,SF,ZF,PF,CF=0)

Der OR-Befehl führt eine logische ODER-Verknüpfung mit den Inhalten des Ziel- und Quelloperanden durch. Das Ergebnis der ODER-Verknüpfung wird im Zieloperanden abgelegt. Der Zieloperand ist ein CPU-Register oder eine Speicherstelle. Der Quelloperand kann ein CPU-Register, eine Speicherstelle oder ein Direktwert sein. Ziel und Quelle müssen die gleiche Größe aufweisen, und nur ein Operand darf für eine Speicherstelle benutzt werden.

Beispiel

```
or ah, 1111110b ; Setze die Bits 7-1 auf 1, Bit-0 bleibt
or ax, bx       ; AX = AX OR BX
or dh, 1        ; Setze niederwertiges Bit im DH- Register auf 1
or Maske, F0h   ; Bits 7-4 auf 1, Bits 3-0 bleiben
```

XOR Ziel, Quelle (OF=0,SF,ZF,AF=?,PF,CF=0)

Der XOR-Befehl verknüpft den Inhalt des Zieloperanden mit dem des Quelloperanden nach den Regeln des Exclusive-Oder. Im Zieloperanden werden die Bits auf 1 gesetzt, die im Quelloperanden einen anderen Zustand aufweisen. Das Ergebnis der XOR-Verknüpfung wird im Zieloperanden abgelegt. Der Zieloperand ist ein CPU-Register oder eine Speicherstelle. Der Quelloperand kann ein CPU-Register, eine Speicherstelle oder ein Direktwert sein. Ziel und Quelle müssen die gleiche Größe aufweisen, es darf nur ein Operand für eine Speicherstelle benutzt werden.

Beispiel

Der XOR-Befehl wird hier für eine individuelle Farbsetzung eines auszugebenden Zeichens mit der entsprechenden Funktion verwendet. Im ersten Schritt laden wird die Farbe Blau (Wert 1) als Attribut für das auszugebende Zeichen in das BL-Register. Im zweiten Schritt vermischen wir die Farbe Blau (Wert 1) mit der Farbe Rot (Wert 4), mit dem Ergebnis Magenta (Wert 5). Im dritten Schritt entfernen wird aus der Farbe Magenta (Wert 5) die Farbe Rot (Wert 4) und erhalten unseren Anfangsfarbwert Blau (Wert 1).

```
mov bl, 1 ; Bitmuster 0000 0001 nach Register bl = Blau
xor bl, 4 ; bl = 0000 0001 XOR0000 0100 = 0000 0101 = Magenta
xor bl, 4 ; bl = 0000 0101 XOR0000 0001 = 0000 0100 = Blau
```

Beim XOR-Befehl resultiert aus einer gleichen Bitkombination immer der Wert 0. Dadurch kann der Befehl auch für das Löschen oder Ausnullen eines Bytes oder Wortes verwendet werden.

```
xor cx, cx ; CX = 0 (mov cx, 0)
xor dl, dl ; DL = 0 (mov dl, 0)
```

Der Vorteil gegenüber einem MOV-Befehl mit dem Wert 0 liegt darin, dass der XOR-Befehl schneller als ein MOV-Befehl ist und weniger Speicherplatz in Anspruch nimmt.

NOT Ziel

Der NOT-Befehl setzt im Zieloperanden alle gelöschten Bits von 0 auf 1 und alle gesetzten Bits von 1 auf 0. Er kehrt also alle Bits des Zieloperanden um. Das Ergebnis der NOT-Verknüpfung wird im Zieloperanden abgelegt. Der Zieloperand ist ein CPU-Register oder eine Speicherstelle.

Beispiel

```
mov ah, 00001111b    ; AH = 00001111
not ah               ; AH = 1111 0000
not ah               ; AH = 0000 1111
```

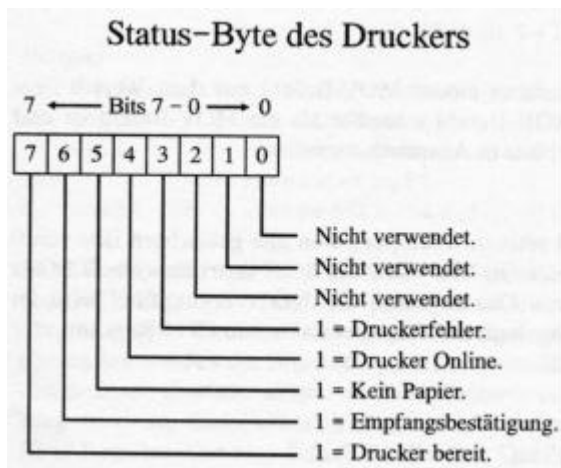
TEST Ziel,Quelle ; Vergleiche Bits (OF=0,SF,ZF,AF=?,PF,CF=0)

Der TEST-Befehl führt intern eine logische UND-Verknüpfung des Zieloperanden mit dem Quelloperanden durch. Das Ergebnis der UND-Verknüpfung wird nicht in einem dieser Operanden abgespeichert. Nur das Statusregister mit seinen Flags wird verändert. Dies hat den Vorteil, dass die einzelnen Bits eines Bytes oder Wortes überprüft werden können, ohne den eigentlichen Wert zu verändern.

Der Zieloperand ist ein CPU-Register oder eine Speicherstelle. Der Quelloperand kann ein CPU-Register, eine Speicherstelle oder ein Direktwert sein. Ziel und Quelle müssen die gleiche Größe aufweisen, es darf nur ein Operand für eine Speicherstelle benutzt werden.

Beispiel

In diesem Beispiel möchten wir den Status eines an einen PC angeschlossenen Druckers abfragen. Der Drucker teilt uns seine Informationen über ein Status-Byte mit. Mit Hilfe dieses Status-Bytes und den darin gesetzten Bits können wir die übergebenen Informationen auswerten.



130.jpg

```
mov ah, 02h          ; Mit diesem Interrupt erfragen wir den Status
xor dx, dx           ; des Druckers . Der Druckerstatus wird anschließend
int 17h              ; im AH-Register zurückgegeben.
...
test ah, 10000000b   ; Bit-7 gesetzt ? (Druckerbereit)
jne Fehler           ; Springe wenn Bit-7 nicht gesetzt ist zu Fehler
test ah, 00100000b   ; Bit-5 gesetzt ? (Kein Papier mehr)
je Fehler            ; Springe wenn Bit-5 gesetzt ist zu Fehler
...
...                  ; <alles ok>
...
Fehler:
...
...                  ; <Fehlerbehandlung>
...
```

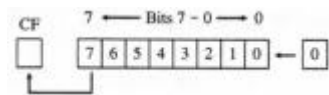
6.3.2 Schiebepfehle

Mit Hilfe der Schiebe- und Rotationsbefehle lässt sich der Inhalt eines Bytes oder Wortes um x Positionen nach links oder rechts verschieben.

SHL Ziel, Quelle (OF,SF,ZF,AF=?,PF,CF)

Der SHL-Befehl bewirkt, dass die Bits im Zieloperanden um x Positionen nach links verschoben werden. Das höherwertige Bit des Zieloperanden wird dabei in das Carry Flag übertragen, die niederwertigen Bits werden mit dem Wert 0 aufgefüllt.

Der Zieloperand ist ein CPU-Register oder eine Speicherstelle. Im Quelloperand steht die Anzahl der Positionen, um die der Zieloperand nach links verschoben werden soll. Soll der Zieloperand nur um ein Bit nach links verschoben werden, so können Sie den Wert 1 angeben. Soll der Zieloperand um mehrere Stellen nach links verschoben werden, muss das CL-Register herangezogen werden.



132.jpg

Beispiel

```
mov ah, 10100101b ; AH = 1010 0101
shl ah, 1          ; AH = 0100 1010 und CF = 1
mov cl, 4         ; CL = 4
shl ah, cl        ; AH = 1010 0000 und CF = 0
```

Der SHL-Befehl kann auch zur Multiplikation eines Operanden mit Zweierpotenzen benutzt werden. Das Verschieben um ein Bit nach links bewirkt im Zieloperanden die Multiplikation mit dem Wert 2, ein nochmaliges Verschieben um ein Bit mit dem Wert 4, ein nochmaliges mit dem Wert 8 usw.

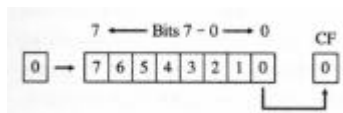
```
mov ah, 10          ; AH = 10
shl ah, 1          ; AH = 20
shl ah, 1          ; AH = 40
shl ah, 1          ; AH = 80
shl ah, 1          ; AH = 160
mov ax, 100        ; AX = 100
mov cl, 3          ; CL = 3
shl ax, cl         ; Verschiebe den Inhalt im AX-Register um drei Bits nach links - was einer Multiplikation
                  ; mit dem Wert acht entspricht - > AX = AX * 8
```

Die Multiplikation mit Hilfe des SHL-Befehl ist etwa 40mal schneller als eine Multiplikation mit dem MUL-Befehl. Wenn Sie eine Multiplikation mit einer Zweierpotenz vornehmen wollen, sollten Sie deshalb den SHL-Befehl bevorzugen.

SHR Ziel, Quelle (OF,SF,ZF,AF=?,PF,CF)

Der Pendant zum SHL-Befehl ist der SHR-Befehl. Der SHR-Befehl bewirkt, dass die Bits im Zieloperanden um x Positionen nach rechts verschoben werden. Das niederwertige Bit des Zieloperanden wird dabei in das Carry-Flag übertragen, die höherwertigen Bits werden mit dem Wert 0 aufgefüllt.

Der Zieloperand ist ein CPU-Register oder eine Speicherstelle. Im Quelloperand steht die Anzahl der Positionen, um die der Zieloperand nach rechts verschoben werden soll. Soll der Zieloperand nur um ein Bit nach rechts verschoben werden, so können Sie den Wert 1 angeben. Soll der Zieloperand um mehrere Stellen nach rechts verschoben werden, so muss das CL-Register herangezogen werden.



133.jpg

Beispiel

```
mov ah, 11111110b ; AH = 1111 1110
shr ah, 1          ; AH = 0111 1111 und CF = 0
mov cl, 4         ; CH = 4
shr ah, cl        ; AH = 0000 0111 und CF = 1
```

Der SHR-Befehl kann auch zur Division mit Zweierpotenzen benutzt werden. Das Verschieben um ein Bit nach rechts bewirkt im Zieloperanden die Division durch den Wert 2, ein nochmaliges Verschieben um ein Bit durch den Wert 4, ein nochmaliges durch den Wert 8 usw.

```

mov ah, 160           ; AH = 160
shr ah, 1            ; AH = 80
shr ah, 1            ; AH = 40
shr ah, 1            ; AH = 20
shr ah, 1            ; AH = 10
mov ah, 64           ; AX = 64
mov cl, 4            ; CL = 4
shr ah, cl           ; Verschiebe den Inhalt im AH-Register um vier Bits nach rechts, was einer Division
                    ; durch den Wert 16 entspricht -> AH = AH / 16

```

Die Division mit Hilfe des SHR-Befehles ist etwa 40mal schneller als eine Division mit dem DIV-Befehl. Wenn Sie eine Division mit einer Zweierpotenz vornehmen wollen, sollten Sie deshalb den SHR-Befehl bevorzugen.

6.4 Sprung- und Schleifenbefehle

Bis jetzt haben wir nur Befehle behandelt, die immer sequentiell und damit der Reihe nach abgearbeitet wurden. Um ein Programm mit einer gewissen Intelligenz auszustatten, benötigen wir noch einige Befehle, damit der Computer auf Entscheidungen reagieren kann. Diese Anweisungen entsprechen den in anderen Sprachen verwendeten IF-THEN-ELSE-Anweisungen.

6.4.1 Sprungbefehle

Mit Hilfe der Sprungbefehle können Sie den Ablauf eines Programms an einer anderen Stelle im Arbeitsspeicher fortführen. Den entsprechenden Befehl finden Sie in fast allen höheren Programmiersprachen in der GOTO-Anweisung. Dieser Befehl ist bei den meisten Programmierern zwar verpönt, in der Assemblerprogrammierung ist er aber die einzige Möglichkeit, gewisse Entscheidungen im Programm zu veranlassen. Die IF-THEN-ELSE-Anweisungen in einer Hochsprache werden natürlich auch durch den Compiler in Sprungbefehle des Assemblers übersetzt. Ein Sprungbefehl ist dabei nichts anderes als ein Ladebefehl auf den Befehlszeiger IP selbst. Erinnern Sie sich noch? Der Befehlszeiger IP zeigt immer auf den nächsten auszuführenden Befehl im Programm. Trifft nun der Assembler auf einen Sprungbefehl, so wird der Inhalt des Befehlszeigers verändert. Die Programmausführung wird an der angegebenen Stelle beim Sprungbefehl weitergeführt. Dies geschieht durch das Erhöhen oder Vermindern des Befehlszeigers durch die Entfernung des Sprungziels in Bytes.

Der unbedingte Sprung - das GOTO in Assembler

Mit diesem Befehl wird das Programm an einer anderen Stelle im Arbeitsspeicher fortgesetzt:

JMP Ziel ;Springe zu der Adresse, die im Zieloperanden steht der JMP-Befehl bewirkt einen Sprung auf eine Adresse im Programm, die mit dem Zieloperanden angegeben wird. Der Zieloperand besteht dabei aus einer Adresse, die meist durch ein Label definiert ist.

Der JMP-Befehl wird im Gegensatz zu den bedingten Sprüngen immer ausgeführt.

Beispiel

```

Addiere: add ax, 10   ; AX = AX + 10
...
jmp Addiere          ; Springe zu dem Label Addiere oder anders:
                    ; Lade den Befehlszeiger (IP) mit der Adresse des Labels Addiere:
...
jmp Ende             ; Springe zu dem Label Ende:
...
Ende:
...
mov ah, 4CH          ; Funktionsparameter (Ende-Programm)
xor al, al           ; AL = 0 -> Returncode zum Betriebssystem
int 21h              ; Funktion des Betriebssystems aufrufen

```

Der JMP-Befehl stellt für die Entfernungsangabe des Sprunges ein zwei Byte großes Feld zur Verfügung, so dass sich das

Sprungziel in einer Entfernung von -32768 bis +32767 Bytes vom Sprungbefehl befinden kann. Das bedeutet, dass mit dem JMP-Befehl jedes Ziel in einem Segment erreicht werden kann.

Bedingte Sprünge - das IF-THEN in Assembler

Um Entscheidungen in Form eines IF-THEN in der Programmiersprache Assembler realisieren zu können, werden Sprungbefehle benötigt, die flexibel auf bestimmte Zustände reagieren. Alle bedingten Sprungbefehle werten dazu das Statusregister des Prozessors aus. Bevor diese Sprungbefehle bestimmte Informationen aus dem Statusregister und den darin befindlichen Statusflags entnehmen können, müssen die Statusflags durch einen vorausgehenden Befehl erst richtig gesetzt werden. Wir benötigen daher einen Vorläufer-Befehl, welchen uns der Assembler mit dem CMP-Befehl zur Verfügung stellt. Der CMP-Befehl vergleicht die angegebenen Operanden miteinander und teilt uns das Ergebnis im Statusregister mit. Die dadurch erreichten Stellungen der Flags im Statusregister können durch die bedingten Sprungbefehle ausgewertet werden, was zu einem Sprung in einen anderen Programmteil führt, wenn die angegebene Bedingung erfüllt ist.

Mit Hilfe des CMP-Befehls können viele Entscheidungen in einem Programm realisiert werden.

CMP Ziel, Quelle ; Vergleiche Ziel mit Quelle (OF,SF,ZF,AF,PF,CF)

Der CMP-Befehl vergleicht den Zieloperanden mit dem Quelloperanden, indem er sie in einem temporären Register voneinander subtrahiert. Beide Operanden bleiben dadurch unverändert. Das Ergebnis der Operation wird im Statusregister gespeichert. Der Zieloperand ist ein CPU-Register oder eine Speicherstelle. Der Quelloperand kann ein CPU-Register, eine Speicherstelle oder ein Direktwert sein. Ziel und Quelle müssen die gleiche Größe aufweisen. Es darf nur ein Operand für eine Speicherstelle benutzt werden.

Beispiel

```
xor ax          ; AX = 0
xor bx          ; BX = 0
cmp ax, bx      ; Vergleiche den Inhalt von AX und BX
                ; AX und BX sind völlig gleich -> Zero-Flag (ZF) = 1

xor bx          ; BX = 0
mov ax, 300     ; AX = 300
cmp ax, bx      ; Vergleiche den Inhalt von AX und BX
                ; AX und BX sind unterschiedlich -> Zero-Flag (ZF) = 0

mov bx, 400     ; BX = 400
cmp ax, bx      ; Vergleiche den Inhalt von AX und BX
                ; AX ist kleiner BX -> Carry-Flag (CF) = 1

mov bx, 200     ; bx = 200
cmp ax, bx      ; Vergleiche den Inhalt von AX und BX
                ; AX ist größer BX -> Carry-Flag (CF) = 0
```

Die durch den CMP-Befehl gesetzten Flags im Statusregister können mit Hilfe der bedingten Sprungbefehle ausgewertet werden. Für die Lösung des Problems *Springe oder springe nicht* stehen Ihnen insgesamt 30 verschiedene Varianten von Verzweigungen zur Verfügung. Da einige Sprungbefehle miteinander identisch sind, schrumpft die Zahl der möglichen Abfragen auf 17 Befehle. In der nachfolgenden Tabelle finden Sie eine Zusammenfassung aller bedingten Sprünge:

Die bedingten Sprunganweisungen des 8086-Prozessors:

Befehl:	Sprung wenn ... Jump if ...	geprüfte Statusflags:
JA/JNBE	größer / nicht-kleiner-gleich above / not-below-equal	CF=0 und ZF=0
JAE/JNB	größer-gleich / nicht-kleiner above-equal / not-below	CF=0
JB/JNAE	kleiner / nicht-größer-gleich below / not-above-equal	CF=1
JBE/JNA	kleiner-gleich / nicht-größer below-equal / not-above	CF=1 oder ZF=1
JE/JZ	gleich / null equal / zero	ZF=1
JG/JNLE	größer / nicht-kleiner-gleich greater / not-less-equal	ZF=0
JGE/JNL	größer-gleich / nicht-kleiner greater-equal / not-less	SF gleich OF
JL/JNGE	kleiner / nicht-größer-gleich less / not-greater-equal	SF ungleich OF
JLE/JNG	kleiner-gleich / nicht-größer less-equal / not-greater	ZF=1
JNE/JNZ	nicht-gleich / nicht-null not-equal / not-zero	ZF=0
JC	Carry Flag (CF) gesetzt Carry	CF=1

137.jpg

JNC	Carry Flag (CF) nicht gesetzt not-Carry	CF=0
JNP/JPO	keine Parität / Parität ungerade not-Parity / Parity-odd	PF=0
JNO	kein Überlauf not-Overflow	OF=0
JNS	kein Vorzeichen (positiv) not-Sign	SF=0
JO	Überlauf Overflow	OF=1
JP/JPE	Parität / Parität-gerade Parity / Parity-even	PF=1

138.jpg

Jxxx Ziel ;Springe, wenn die Bedingung erfüllt ist, zu Ziel

Jede der vorgestellten Sprunganweisungen testet die Flags im Statusregister und führt einen Sprung zum Zieloperanden aus, wenn die angegebene Bedingung erfüllt ist. Dazu wird der Zieloperand geladen, je nach Vor- oder Rückwärtsverzweigung vorzeichenerweitert, und zum Befehlszeiger IP addiert. Andernfalls wird der nächste Befehl hinter dem Sprungbefehl verarbeitet. Der Zieloperand besteht dabei aus einer Adresse, die häufig durch ein Label definiert ist.

Beispiel

; <Wir sind im Datensegment>

...
Meld_gleich db "Operanden sind gleich groß", "\$ "

```

Meld_groess      db      "Erster Operand ist größer" , "$"
Meld_klein       db      "Erster Operand ist kleiner" , "$"
Meld_fehler      db      "Überlauf nach arithm. Operation" , "$"

```

```

; <Wir sind im Codesegment>

```

Anfang:

```

    cmp ax, bx          ; Vergleiche die Inhalte von AX und BX
    je  Gleich          ; Springe wenn AX = BX zu Gleich:
    jg  Groesser        ; Springe wenn AX > BX zu Groesser:
    jl  Kleiner         ; Springe wenn AX < BX zu Kleiner:
    jmp Anfang          ; Springe ohne wenn und aber zu Anfang:
    ...
    cmp Zaehler,100     ; Vergleiche den Inhalt von Zaehler mit 100
    jg  Groesser        ; IF Zaehler > 100 THEN gehe zu Groesser:
    jmp Kleiner         ; ELSE gehe zu dem Label Kleiner:
    ...
    mul bh              ; Auch der MUL-Befehl verändert die Flags
    jo  Fehler          ; Ergebnis kann im AX-Register nicht mehr dargestellt werden -> Springe zu Fehler:
    ...

```

Gleich:

```

    lea dx, Meld_gleich ; String, der ausgegeben werden soll
    jmp String_aus       ; String über Funktion 09h Interrupt 21h ausgeben

```

Groesser:

```

    lea dx, Meld_gross
    jmp String_aus

```

Kleiner:

```

    lea dx, Meld_klein
    jmp String_aus

```

Fehler:

```

    lea dx, Meld_fehler
    jmp String_aus

```

String_aus:

```

    mov ah, 09h         ; Ausgabe der Zeichenkette auf die
    int 21h             ; das DX-Register zeigt
    jmp Anfang          ; und wieder zurück zu Anfang

```

Ein Wermutstropfen zum Schluss!

Im Gegensatz zum unbedingten Sprungbefehl *JMP* verfügen die bedingten Sprungbefehle nur über ein 1 Byte großes Feld, um die Entfernung der Zieladresse aufzunehmen. Das bedeutet, dass man mit den bedingten Sprungbefehlen nur die Labels erreicht, die innerhalb einer Reichweite von -128 oder +127 Bytes liegen. Folgendes Beispiel würde der Assembler mit einer Fehlermeldung quittieren:

```

...
weit_weg:
...
... ; <viele - viele Befehle>
...
dec ah          ; 200 Bytes später -> AH = AH - 1
jz weit_weg     ; Error -> Sprungziel ist zu weit weg
mov ah, 100     ; AH = 100
...

```

Befindet sich ein Sprungziel bei der Verwendung eines bedingten Sprunges außerhalb der Bytegrenze von -128 oder + 127 Bytes, so wird der Assembler bei der Assemblierung damit nicht einverstanden sein. Da wir den Assembler aber auch nicht unnötig reizen wollen, werden wir den Sprung mit Hilfe des *JMP*-Befehls realisieren. Wie Sie wissen, kann der *JMP*-Befehl jedes Sprungziel innerhalb eines Segmentes erreichen. Durch das Negieren des bedingten Sprunges von *jz* in *jnz* und den Einsatz des *JMP*-Befehls können wir den Befehl in unserem obigen Beispiel realisieren:

weit_weg:

```

...
dec ah          ; AH = AH - 1
jnz weiter_1    ; IF AH > 0 THEN springe zu weiter_1:

```

```

        jmp weit_weg          ; IF AH = 0 THEN springe zu weit_weg:
weiter_l:
        mov ah, 100          ; AH =100
        ...

```

Jetzt ist der Assembler endgültig zufrieden und wird die Sequenz in den Maschinencode umwandeln. Dieses Entfernungsproblem bei bedingten Sprungbefehlen tritt häufig in größeren Programmen auf. Wenn Sie beim Label `weiter_1999` angekommen sind, könnte das Programm durch die vielen Sprungadressen doch etwas unübersichtlich werden. Abhilfe schafft hier der `$`-Operator. Da der `JMP`-Befehl fünf Bytes Maschinencode benötigt, kann das vorhergehende Beispiel auch ohne die Verwendung von `weiter_...` realisiert werden.

```

        ...
weiter_weg:
        ...
        dec ah
        jnz $ + 5            ; Springe zu mov ah, 100 1.)
        jmp weit_weg        ; Eigentliches Sprungziel
        mov ah, 100         ; < --- 1.)

```

Das Dilemma mit der Sprungentfernung trifft auch für Schleifen-Befehle zu. Daher ist dieser kleine Trick mit dem `$`-Operator auch in vielen Programmen anzutreffen. Eine derartige Eskapade sollten Sie in einem Programm aber genau dokumentieren, da das Einfügen eines weiteren Befehls **nach** der Anweisung `jnz $ + 5` böse Folgen für den weiteren Programmablauf hinterlässt.

6.4.2 Schleifenbefehle

Schleifen werden gebildet, um eine Sequenz von Befehlen mehrfach auszuführen. In der Programmierung in Assembler werden Schleifen wieder mit bedingten Sprunganweisungen erstellt. Am Anfang einer Schleife benötigen wir hierzu ein Label, welches den Kopf der Schleife darstellt. Danach folgt die Befehlssequenz, die durch die Schleife mehrfach wiederholt werden soll. Das Ende der Schleife wird durch den Schleifenbefehl festgelegt. Der 8086-Prozessor stellt für die Konstruktion von Schleifen vier bedingte Sprunganweisungen zur Verfügung, die wir uns jetzt etwas näher anschauen wollen.

LOOP Ziel ;Verzweige so lange zu Ziel, bis CX-Register = 0
Der `LOOP`-Befehl verzweigt so lange zu dem Zieloperanden, bis das CX-Register den Wert null aufweist. Der `LOOP`-Befehl vermindert dazu bei jedem Aufruf das CX-Register um den Wert 1 und kontrolliert anschließend dessen Inhalt. Wenn das CX-Register den Wert null aufweist, wird der nächste Befehl nach der `LOOP`-Anweisung verarbeitet. Ist das CX-Register ungleich null, so wird zu dem beim `LOOP`-Befehl angegebenen Ziel verzweigt und die Befehlssequenz erneut durchlaufen. Das Statusregister wird durch den `LOOP`-Befehl nicht verändert. Dieser Befehl gehört wie die restlichen Schleifenbefehle zur Gruppe der bedingten Sprungbefehle, so dass sich der Zieloperand in einer Reichweite von -128 bis +127 Bytes vom `LOOP`-Befehl befinden muss.

Beispiel

Als ersten Einstieg wollen wir das Zeichen * zweitausendmal auf dem Bildschirm ausgeben:

```

; <Wir sind imCodesegment>

        mov cx, 2000        ; Lade CX mit dem Schleifenzähler
        mov dl, "*"         ; Funktion erwartet das Zeichen in DL
        mov ah, 2           ; Funktion zur Ausgabe eines Zeichen
Schleife: < ----- 1.)
        int 21h             ; Funktion aufrufen I
        loop Schleife       ; solange CX nicht 0 ist, springe zu Schleife: 1.)

```

In nächsten Beispiel möchten wir eine Tabelle in Form einer Zeichenkette nach dem Buchstaben 0 durchsuchen. Wird der Buchstabe gefunden, so soll das AH-Register um den Wert 1 erhöht werden.

```

; <Wir sind im Datensegment>
Tabelle db "Ein Beispiel zu dem LOOP-Befehl !"
Tab_laeng EQU $ - Tabelle ; Länge der Tabelle

; <Wir sind im Codesegment>
        mov cx, Tab_laeng   ; Lade CX mit dem Schleifenzähler
        lea bx, Tabelle     ; Zeiger auf das erste Zeichen der Tabelle
        xor ah, ah         ; AH = 0

```

```

        mov  dh, „O“          ; Vergleichswert nach DH
A_Schleife:
        cmp  dh, [bx]         ; Vergleiche auf Buchstaben O
        jne  E_Schleife       ; Kein O - springe zu E_Schleife:
        inc  bx                ; O gefunden - AH um 1 erhöhen
E_Schleife:
        inc  bx                ; Nächstes Zeichen der Tabelle
        loop A_Schleife       ; solange CX nicht 0 - springe zu A_Schleife:
        ...

```

Der LOOP-Befehl kann auch mit den beiden nachfolgenden Befehlen realisiert werden:

```

        ...
        dec  cx                ; CX = CX - 1
        jnz  A_Schleife       ; Solange Ergebnis nicht 0 springe zu A_Schleife:
        ...

```

Der LOOP-Befehl ist um ein vielfaches schneller und benötigt weniger Platz im Maschinencode als die obige Befehlssequenz. Mit dem LOOP-Befehl lassen sich die meisten Schleifenkonstruktionen realisieren. Es gibt aber Schleifenkonstruktionen, denen der LOOP-Befehl nicht mehr ausreicht. Des öfteren wird beispielsweise eine Abbruchbedingung benötigt, die nicht nur das CX-Register und damit das Ende der Schleife kontrolliert, sondern zusätzlich auch noch das Zero-Flag mitüberprüft. Diesen Service bieten die beiden Befehle LOOPE und LOOPNE.

Die Befehle LOOPE und LOOPNE

Beide Befehle arbeiten ähnlich wie der LOOP-Befehl und führen einen bedingten Sprung nach Ziel so lange durch, bis das CX-Register den Wert null aufweist. Dies erfolgt unter der Berücksichtigung des Zero Flags. Eine Schleife wird dann so lange wiederholt, bis

- das CX-Register den Wert 0 hat
- oder
- das Zero Flag gelöscht (ZF=0) ist (LOOPE)
- das Zero Flag gesetzt (ZF=1) ist (LOOPNE)

Beispiel zu LOOPE

Suche in einer Tabelle ein Zeichen ungleich *.

Beende die Suche, wenn ein Zeichen ungleich * gefunden wird oder das Ende der Tabelle erreicht ist.

```

;<Wir sind im Datensegment>
Tabelle      db      "*****?*****"
Tab_laeng    EQU     $ - Tabelle      ; Länge der Tabelle

;<Wir sind im Codesegment>
...
        mov  cx, Tab_laeng        ; Länge der Tabelle
        mov  bx, -1              ; BX = Index
Schleife:
        inc  bx                  ; Ein Element der Tabelle weiter
        cmp  Tabelle[bx], "*"     ; vergleiche auf "*" - wenn gleich, wird das Zero Flag auf 1 gesetzt
        loope Schleife           ; Ende der Schleife, wenn CX = 0 oder ZF = 0
        ...

```

Beispiel zu LOOPNE

Der Inhalt einer Tabelle wird so lange verarbeitet, bis ein ? erkannt wird oder die Tabelle zu Ende ist.

```

;<Wir sind im Datensegment>
...
Tabelle      db      "Sprung ? oder nicht Sprung"
Tab_laeng    EQU     $ - Tabelle      ; Länge der Tabelle

;<Wir sind im Codesegment>
...
        mov  cx, Tab_laeng        ; Lade CX mit dem Schleifenzähler
        lea  bx, Tabelle          ; das erste Zeichen der Tabelle
        dec  bx                  ; wegen inc bx nach Schleife:

```

```

    mov dh, "?"                ; Vergleichswert nach DH
Schleife:
    inc bx                    ; Nächstes Zeichen der Tabelle
    cmp dh, [bx]             ; vergleiche auf "?" - wenn gleich, wird das Zero Flag auf 1 gesetzt
    loopne Schleife         ; Ende der Schleife, wenn CX = 0 oder ZF = 1
    ...

```

Nachteilig ist, dass die Befehlssequenz in der Schleife immer einmal durchlaufen werden muss, da die Endebedingung erst am Schluss überprüft wird. Für diesen Fall gibt es jedoch den JCXZ-Befehl.

Der JCXZ-Befehl

Der JCXZ-Befehl verzweigt zu der angegebenen Adresse, wenn der Inhalt des CX-Registers 0 ist. Ist das CX-Register nicht 0, so wird der nächste Befehl, der dem JCXZ-Befehl folgt, ausgeführt.

Auf diese Weise können Sie das CX-Register vor einer Schleife auf einen gültigen Wert hin überprüfen und verhindern, dass eine Schleife mit einem Nullwert im CX-Register begonnen wird.

Beispiel

```

; <Wir sind im Codesegment>
    JCXZ CX_ist_null        ; Wenn CX = 0, springe zu CX_ist_null: -----> 1.)
Schleife:
    ...
; <Befehle>
    ...
    loop Schleife          ; CX = CX - 1 - wenn CX nicht 0, dann zu Schleife: --- > 2.)
CX_ist_null:
    ; <-----> 1.)

```

Die Schleife wird im obigen Beispiel nur ausgeführt, wenn der Inhalt des CX-Registers einen Wert ungleich null aufweist. Dieser Befehl wird also aus Sicherheitsgründen verwendet. Würde die Schleife mit einem Wert gleich null im CX-Register begonnen, so wird die Schleife genau 65535mal ausgeführt. Denn bevor der LOOP-Befehl zu der angegebenen Sprungadresse verzweigt, wird zuerst das CX-Register um den Wert eins vermindert. Null minus eins ergibt dabei aber nicht den Wert -1, sondern den hexadezimalen Wert FFFFh, in dezimaler Schreibweise 65535.

Wird der Schleifenzähler nicht unmittelbar vor den LOOP-Befehl gesetzt, so sollte aus Sicherheitsgründen immer eine Prüfung des CX-Registers auf einen Wert ungleich null durchgeführt werden.

6.5 String befehle

In der täglichen Programmierpraxis werden häufig Befehle benötigt, die ganze Zeichenketten in Form von Tabellen und Datenpuffern verarbeiten. Der 8086-Prozessor stellt für das Kopieren, Verschieben und Durchsuchen von Zeichenketten fünf Grundbefehle zur Verfügung:

- LODSx	Laden einer Zeichenkette
- STOSx	Schreiben in eine Zeichenkette
- MOVSx	Verschieben einer Zeichenkette
- SCASx	Durchsuchen einer Zeichenkette
- CMPSx	Vergleichen zweier Zeichenketten

Byte- oder Wort-Größe

Dabei kann der Assembler aber nicht wissen, ob er die Zeichenkette byte- oder wortweise verarbeiten soll. Um den Assembler über die Verarbeitungsweise zu informieren, müssen Sie dies bei den obigen Befehlen kenntlich machen. Durch das Ersetzen des nachgestellten x durch eine B für Byte oder durch ein W für Wort teilen Sie dem Assembler die gewünschte Verarbeitungsform mit.

Sie können aber auch das Label der Zeichenkette angeben, die verarbeitet werden soll.

```

; <Wir sind im Datensegment>
...
String      db      "ABCDEFGH"
...

```

```
; <Wir sind im Codesegment>
```

```
...  
lea si, String          ; Quelle  
loadsw                 ; Wort-Größe  
loadsb                 ; Byte-Größe  
lods String          ; Byte-Größe  
...
```

Ziel- und Quelloperand

Bei allen Stringbefehlen sind folgende Regeln zu beachten:

- Der Quelloperand wird durch das Registerpaar DS:SI adressiert.
- Der Zieloperand wird durch das Registerpaar ES:DI adressiert.
- Die Register SI und DI werden bei der Ausführung der Stringbefehle automatisch erhöht oder verringert.
- Werden Stringbefehle mehrmals wiederholt, wird das CX-Register als Zähler verwendet.

Von links nach rechts oder umgekehrt

Der Prozessor verfügt im Statusregister über ein Direction Flag DF. Ist dieses Flag (DF=0) gelöscht, so wird die Zeichenkette in aufsteigender Richtung von links nach rechts verarbeitet.

Möchten Sie die Zeichenkette in absteigender Richtung von rechts nach links abarbeiten, so müssen Sie das Direction Flag (DF=1) setzen. Verwenden Sie hierzu folgende Befehle:

```
CLD                    ; lösche Direction Flag / links nach rechts (DF=0)  
STD                    ; setze Direction Flag / rechts nach links (DF=1)
```

Nach diesem Einstieg in die Stringbefehle wollen wir jeden einzelnen Befehl näher anschauen und anhand eines Beispiels vertiefen.

Stringbefehle zum Kopieren von Speicherblöcken

LODSx ;lade den Akkumulator mit dem Inhalt von DS:SI

Der LODS-Befehl LOAD String überträgt ein Byte (LODSB) oder ein Wort [LODSW] aus dem durch das Registerpaar DS:SI adressierten Speicherbereich in den Akkumulator. Mit LODSB wird ein Byte des durch DS:SI adressierten Speicherblocks in das AL-Register eingelesen, durch LODSW ein Wort in das AX-Register. Das SI-Register wird dabei je nach Direction-Flag automatisch erhöht oder vermindert, bei einer Byteoperation um den Wert eins, bei einer Wortoperation um den Wert zwei. Der LODS-Befehl geht dabei standardmäßig auf die Adresse, die durch das Registerpaar DS:SI adressiert wird. Bevor Sie den LODS-Befehl verwenden, müssen Sie deshalb zuerst die zu verarbeitende Zeichenkette in das SI-Register laden.

Beispiel

Der Inhalt einer Tabelle wird so lange verarbeitet, bis ein ? erkannt wird oder die Tabelle zu Ende ist.

```
; <Wir sind im Datensegment>  
String      db      "12?34"  
String_1    equ     $ - String      ; Länge der Zeichenkette  
...  
; <Wir sind im Codesegment>  
...  
mov cx, String_1      ; Lade CX mit dem Schleifenzähler  
lea si, String        ; SI zeigt auf "1" in String  
cld                  ; aufsteigende Richtung  
Schleife:  
lods String          ; kopiere ein Byte von DS:SI -> AL  
cmp al, "?"          ; vergleiche AL mit "?"  
loopne Schleife     ; Ende, wenn al = "?" oder CX = 0
```

STOSx ;kopiere den Inhalt des Akkumulators nach ES:DI

Der STOS-Befehl STORE String überträgt ein Byte [STOSB] oder ein Wort [STOSW] aus dem Akkumulator in den durch das Registerpaar ES:DI adressierten Speicherbereich. Mit STOSB wird der Inhalt des AL-Registers in den durch das Registerpaar ES:DI adressierten Speicherblock kopiert, mit STOSW der Inhalt des AX-Registers.

Der STOS-Befehl arbeitet also genau umgekehrt wie der LODS-Befehl. Zum Unterschied verwendet der STOS-Befehl das Registerpaar ES:DI als Zieladresse der Datenübertragung. Das DI-Register wird dabei je nach Direction Flag automatisch erhöht oder vermindert: bei einer Byteoperation um den Wert eins und bei einer Wortoperation um den Wert zwei.

Beispiel

In diesem Beispiel möchten wir die zwei Zeichenketten *Lw* und *Dir* nach *Pfad* kopieren. *Pfad* hat dann den Inhalt *LW:Directory*.

```

; <Wir sind im Datensegment>
Lw      db      " LW:"                ; Feld für das Laufwerk
Lw_1    EQU     $ - Lw                ; Länge von Lw
Dir      db      "Directory"          ; Feld für das Directory
Dir_1    EQU     $ - Dir              ; und wieder die Länge des Feldes
Pfad     db      Lw_1 + Dir_1         DUP (?); Feld für die Pfadangabe
...
; <Wir sind im Codesegment>
...
cld                    ; aufsteigende Richtung
mov ax, ds             ; Datensegment zwischenspeichern
mov es, ax            ; Extrasegment zeigt jetzt auf auf das Datensegment (STOSx)
Laufwerk:
lea si, Lw             ; Quelle - was wird kopiert
lea di, Pfad          ; Ziel - wohin wird kopiert
mov cx, Lw_1          ; Schleifenzähler=Länge von Lw
mov dh, 1             ; Schalter
Schleife:
lodsb                 ; Zeichen aus Quelle holen (Byte)
stosb                 ; Zeichen nach Ziel kopieren (Byte)
loop Schleife         ;
cmp dh, 0             ; Ende der Verarbeitung ?
je Ende              ; Ja - dann springe zu Ende:
Directory:
mov cx, Dir_1         ; Dir noch an Pfad anhängen
lea si, Dir           ; neue Quelle - was kopiert werden soll
mov dh, 0             ; danach ist dann Schluss - Schalter = 0
jmp Schleife         ; noch mal kopieren
Ende:

```

MOVSB; Kopiere den Inhalt der Speicherstelle DS:SI in die Adresse ES:DI

Der MOVSB-Befehl MOVE String ist eine Kombination aus den beiden Befehlen LODS und STOS. Zum Unterschied wird der zu kopierende Byte- oder Wort-Wert von DS:SI (Quelle) nach ES:DI (Ziel) nicht im Akkumulator zwischengespeichert. Der Inhalt des AX-Registers bleibt damit durch den MOVSB-Befehl unverändert.

Beispiel

Eine Zeichenkette soll byteweise kopiert werden.

```

; <Wir sind im Datensegment >
Quelle  db      "Ein String der kopiert werden soll"
Quelle_1 EQU     $ - Quelle            ; Länge des Strings Quelle
Ziel    db      Quelle_1 DUP (?)      ; Zielfeld mit Länge von Quelle
...
<Wir sind im Codesegment>
...
mov ax, ds            ; Datensegment Zwischenspeichern
mov es, ax           ; ES zeigt jetzt auch auf DS
lea si, Quelle        ; Quelle - was wird kopiert
lea di, Ziel          ; Ziel - wohin wird kopiert
mov cx, Quelle_1     ; Schleifenzähler = Länge von Quelle
cld                  ; aufsteigende Richtung
Kopiere:
movsb                 ;Quelle (DS:SI) --> Ziel (ES:DI)
loop Kopiere
...

```

Stringoperationen mit dem REP-Befehl wiederholen

Um einen Speicherbereich in voller Länge zu verarbeiten, mussten wir bisher immer den LOOP-Befehl bemühen. Es geht aber auch anders. Der Prozessor stellt für die automatische Wiederholung von Stringfunktionen ein Wiederholungspräfix in Form des REP-Befehles REPEAT string zur Verfügung. Der REP-Befehl wird vor den eigentlichen Stringbefehl gestellt. Er bewirkt, dass der Stringbefehl nach dem Wiederholungspräfix REP so lange ausgeführt wird, bis das CX-Register den Wert null erreicht hat. Steht das CX-Register bereits auf null, so wird der Befehl nicht ausgeführt.

Beispiel

Der LOOP-Befehl im letzten Beispiel

```
...
mov cx, Quelle_1      ; Schleifenzähler = Länge von Quelle
Kopiere :             ; <-----+
movsb                 ; Quelle (DS:SI) --> Ziel (ES:DI)
loop Kopiere          ; -----+
...
```

könnte durch den Einsatz des REP-Befehls wie nachfolgend beschrieben ersetzt werden:

```
...
mov cx, Quelle_1      ; Schleifenzähler = Länge von Quelle
rep movsb             ; Quelle (DS:SI) --> Ziel (ES:DI)
...
```

Der REP-Befehl arbeitet dabei ähnlich wie der LOOP-Befehl. Nach jeder Ausführung eines Stringbefehls wird das CX-Register um den Wert 1 vermindert. Solange das CX-Register einen Wert ungleich null aufweist, wird der Stringbefehl wiederholt. Der Vorteil ist, dass der REP-Befehl erheblich schneller arbeitet als der LOOP-Befehl.

Wie für den LOOP-Befehl existieren für den REP-Befehl noch zwei weitere Varianten, nämlich REPE (REPeat while Equal) und REPNE (REPeat while Not Equal). Diese beiden Befehle sind besonders bei den Stringbefehlen interessant, die Zeichenketten durchsuchen oder vergleichen. Damit sind wir schon beim nächsten Thema.

Stringbefehle zum Durchsuchen von Speicherblöcken

SCASx ;Vergleiche den Inhalt des Akkumulators mit der Speicherstelle ES:DI (OF,SF,ZF,AF,PF,CF)

Der SCAS-Befehl vergleicht den Inhalt der Speicherstelle ES:DI mit dem Akkumulator. Wird mit einer Byte-Größe [SCASB] gearbeitet, so wird das Zeichen im AL-Register mit der Speicherstelle ES:DI (Byte) verglichen, bei einer Wort-Größe (SCASW) wird der Inhalt des AX-Registers mit dem Inhalt ES:DI (Wort) verglichen. Dabei wird der Inhalt des Akkumulators vom Registerpaar ES:DI adressierten Operanden subtrahiert, das Ergebnis wird nicht abgespeichert. Die Informationen über den Vergleich werden wie beim CMP-Befehl im Statusregister angezeigt und können durch bedingte Sprunganweisungen oder die Befehle REPE und REPNE ausgewertet werden.

Beispiel

Der Inhalt einer Zeichenkette wird so lange verarbeitet, bis ein Z erkannt wird oder die Zeichenkette zu Ende ist.

```
; <Wir sind im Datensegment>
Ziel db "Ein Zielfeld das durchsucht werden soll"
Ziel_1 EQU $ - Ziel ; Länge von Ziel
...
; <Wir sind im Codesegment>
...
mov ax, ds ; Datensegment Zwischenspeichern
mov es, ax ; ES zeigt jetzt auch auf DS
mov cx, Ziel_1 ; lade CX mit dem Schleifenzähler
lea di, Ziel ; Zeiger auf den Anfang des Strings Ziel
mov al, "Z" ; Vergleichswert nach AL
cld ; auf steigende Richtung
repne scasb ; Ende, wenn ES : DI = "Z" oder CX = 0
cmp cx, 0 ; Vergleiche CX auf null
je nicht_gef ; wenn CX = 0 - kein "Z" gefunden
...
```

Nachdem der SCAS-Befehl das Zeichen im AL-Register mit der durch das Registerpaar ES:DI adressierten Speicherstelle verglichen hat, wertet SCAS den Vergleich aus und setzt das Statusregister entsprechend. Das angegebene Wiederholungspräfix REPNE (REPeat while Not Equal) überprüft nun ebenfalls die Flags im Statusregister. Ist das gesuchte Zeichen im AL-Register ungleich mit dem durch ES:DI adressierten Zeichen, so wird der SCAS-Befehl erneut wiederholt, bis das Zeichen in AL mit ES:DI identisch ist oder das CX-Register den Wert null aufweist. Wird eine Übereinstimmung mit dem Inhalt des AL-Registers und der Speicherstelle (ES:DI) festgestellt, so wird der nächste Befehl nach dem Stringbefehl ausgeführt. In unserem Beispiel würde nach der fünften Wiederholung von SCAS eine Übereinstimmung der Zeichen festgestellt und die Verarbeitung abgebrochen. Das DI-Register wird nach jedem Vergleich mit SCAS je nach Direction-Flag (DF) erhöht oder gesenkt. Da wir mit dem CLD-Kommando eine aufsteigende Verarbeitungsrichtung gewählt haben, wurde DI nach dem SCAS-Befehl um den Wert

eins (5CA5ß=Byte-Größe) erhöht und anschließend der Vergleich mit REPNE durchgeführt. Das DI-Register zeigt somit nicht auf die Speicherstelle, an der das Zeichen gefunden wurde, sondern auf das nächste Zeichen ;'. Möchten Sie das gefundene Zeichen bearbeiten, müssen Sie DI entsprechend korrigieren.

CMPSx ; vergleiche den Inhalt von Quelle (DS:SI) mit dem Inhalt von Ziel (ES:DI) (OF,SF,ZF,AF,PF,CF)

Der CMPS-Befehl COMPUse 5tring vergleicht den Inhalt des Registerpaars DS:SI mit dem Inhalt der durch ES:DI adressierten Speicherstelle. Durch das Subtrahieren des Inhaltes des Zielperanden ES:DI vom Inhalt des Quelloperanden DS:SI wird das Ergebnis des Vergleiches im Statusregister abgespeichert.

Wie beim SCAS-Befehl kann der Zustand des Statusregisters durch einen bedingten Sprung oder die Befehle REPE und REPNE ausgewertet werden.

Beispiel

; Zwei Zeichenketten sollen auf Gleichheit überprüft werden.

; <Wir sind im Datensegment>

```
...
Quelle db "Gleich oder nicht gleich ? "
Ziel db "Gleich Oder nicht gleich ? "
Ziel_l EQU $ - Ziel ; Länge von Ziel
...
```

; <Wir sind im Codesegment>

```
...
mov ax, ds ; Datensegment Zwischenspeichern
mov es, ax ; ES zeigt jetzt auch auf DS
mov cx, Ziel_l ; lade CX mit dem Schleifenzähler
lea si, Quelle ; Zeiger auf den Anfang von Quelle
lea di, Ziel ; Zeiger auf den Anfang von Ziel
cld ; aufsteigende Richtung
repe cmpsb ; Ende., wenn DS : SI ungleich ES : DI oder CX = 0 = Ende des Strings
je Strings_sind_gleich
```

Strings_sind_gleich:

...

6.6 Pointer-Angabe

Bei allen Maschinenbefehlen müssen die verwendeten Operanden die gleiche Größe besitzen. Kann der Assembler die Größe eines Operanden nicht erkennen, erzeugt er eine Warnung oder Fehlermeldung. Hier ist der PTR-Operator erforderlich.

```
--- byte ptr ----
Befehl --- ----- Operand
--- word ptr ---
```

Der PTR-Operator wird verwendet, um den Datentyp einer Variablen oder eines Labels festzulegen. In der Programmiersprache Assembler arbeiten wir meistens mit Byte- und Wort-Werten. In vielen Fällen legen die Operanden des Befehls den Datentyp Byte oder Wort fest. Welche Datentypen verarbeitet werden sollen, ist bei den nachfolgenden Befehlen durch den Assembler eindeutig zu erkennen:

```
mov ax, bx ; wort
mov [bx], cl ; byte
mov al, 200 ; byte
inc cx ; wort
dec al ; byte
```

Bei den nächsten Befehlen ist es dem Assembler nicht möglich, den richtigen Datentyp zu bestimmen, und er gibt deshalb eine Fehlermeldung aus.

```
mov [bx], 255 ; <--- Error - Operand must have size
inc [bx] ; <--- " "
```

Der Assembler hat keine Möglichkeit festzustellen, ob bei der durch das BX-Register adressierten Speicherstelle ein Byte- oder

Wort-Wert verändert werden soll. In diesem Fall ist es notwendig, dem Assembler eine Hilfestellung zu geben. Mit dem PTR-Operator teilen Sie dem Assembler mit, welcher Datentyp verarbeitet wird. Die Angabe BYTE PTR gibt dem Assembler die Information, dass der Operand als Byte betrachtet wird. WORD PTR informiert darüber, dass es sich bei dem Operanden um ein Wort handelt.

```
; Schiebe den Wert 255 in das Byte, auf das BX zeigt
    mov [bx], byte ptr 255    ; byte

; Erhöhe das Wort, auf das BX zeigt, um den Wert 1
    inc word ptr [bx]        ; wort
```

6.7 Übung 6

Aufgabe

Schreiben Sie ein Programm, das alle Großbuchstaben eines Satzes in Kleinbuchstaben umwandelt. Großbuchstaben liegen im dezimalen Wertebereich des ASCII-Codes von 65 bis 90, Kleinbuchstaben haben den Wertebereich 97 bis 122. Wie Sie aus der nachfolgenden Tabelle ersehen können, haben Großbuchstaben und Kleinbuchstaben im ASCII-Code einen Wertunterschied von 32.

Zeichen	ASCII-Code	Differenz	ASCII-Code	Zeichen
A	65	32	97	a
B	66	32	98	b
C	67	32	99	c
...
X	88	32	120	x
Y	89	32	121	y
Z	90	32	122	z

6.8 Lösung 6

```
*****
;
; Programm          : Umw.asm
; Funktion          : Umwandeln von Groß- in Kleinbuchstaben
; Assemblierung     : Tasm Umw (Borland) oder Masm Umw (Microsoft)
; Linken            : Tlink Umw " " oder Link Umw " "
*****

; ***** Konstanten *****
Lowb equ "A"          ; Anfangsbereich Großbuchstaben ="A"= 65
Highb equ "Z"         ; Endebereich Großbuchstaben ="Z"= 90
Differenz equ "a" - "A" ; Differenz zwischen Groß- und Kleinbuchstaben -> 97 - 65 = 32

DATEN          SEGMENT          ; hier gehören die Daten hin
Satz           db "ABCDEFGHJI"  ; in Kleinbuchstaben umwandeln
Laenge         equ $ - Satz     ; $ steht für die aktuelle Position
; $ - Satz = Länge des Satzes (Byte)
Endesatz       db "$"          ; Ende der Zeichenkette für die Ausgabe
DATEN          ENDS

STAPEL         SEGMENT          ; Stack mit 256 Byte reserviert
db 256 dup (?)
STAPEL         ENDS

CODE           SEGMENT          ; ab hier übernehmen wir das Kommando
ASSUME CS:CODE, DS :DATEN, SS :STAPEL
; Segmente zuweisen

Anfang:
    mov ax, DATEN             ;
    mov ds, ax               ; Datensegment zuweisen
    lea bx, Satz              ; Offset-Adresse des Satzes nach BX
    mov cx, Laenge           ; Laenge des Satzes als Schleifenzaehler
```

```

Pruef :
    cmp [bx], byte ptr Lowb ; vergleiche auf Bereichsanfang
    jl lesen                ; wenn kleiner weiterlesen - kein Großbuchstabe
    cmp [bx], byte ptr Highb ; vergleiche auf Bereichsende
    jg lesen                ; wenn größer weiterlesen - kein Großbuchstabe

; Großbuchstabe wurde gefunden - durch Addieren des Differenzbetrages auf der durch
; das BX-Register adressierten Speicherstelle - Groß- nach Kleinbuchstaben umwandeln

    add [bx], byte ptr Differenz
lesen:
    inc bx                  ; erhöhe BX um den Wert 1 -> BX zeigt dann auf das
                           ; nächste Element (Buchstaben) des Satzes
.schleife:
    loop pruef             ; Solange CX nicht 0 - springe zu pruef:
    lea dx, Satz           ; damit Sie sehen, dass die Umwandlung
    mov ah, 09h           ; funktioniert, wird der String über den
    int 21h               ; Interrupt 21h ausgegeben.
Endeprg:
    mov ah, 4Ch           ; Ende des Programms, und
    xor al, al           ; Returncode zu MS-DOS ist null
    int 21h             ; zurück zum Betriebssystem
CODE    ENDS
        END Anfang      ; beginne beim Label Anfang:

```