

Vorlesung in Assembler-Programmierung

I. Grundlagen und Einsatz der Assembler-Programmierung

0. Verwendete Literatur:

Backer, Reiner: Assembler: Maschinennahes Programmieren, Reinbek bei Hamburg, 2. Auflage 2003, ISBN 3 499 61224 0

⇒ **Einführungs-Lehrbuch; auf 8086 beschränkt; viele gute einfache Beispiele**

Link, Wolfgang: Assembler Programmierung, 12. Auflage 2006, ISBN 3-7723-7798-X

⇒ **Seit Jahren Standard-Buch für 8086-Programmierung; gut verständlich, viele Beispiele; alphabetische Befehlsbeschreibungen vom 8086 bis Pentium im Anhang**

Norton, Peter et al.: Peter Norton's Assemblerbuch, Haar bei München 1988, ISBN 3-89090-624-9

⇒ **Didaktisch sehr gut aufbereitete Einführung in die Maschinensprache-Programmierung des „Altmeisters“**

Rohde, Joachim et al.: Assembler: Grundlagen der Programmierung, 2. Aufl., Heidelberg 2006, ISBN 3-8266-1469-0

⇒ **Bezieht gleichermaßen 32-Bit-Programmierung mit MMX-Technologie mit ein; umfassendes Werk (eher für Fortgeschrittene geeignet)**

Rohde, Joachim: Assembler: Ge-packt: Alle Befehlssätze von 8086 bis Pentium 4, Bonn 2001, ISBN 3-8266-0786-4

⇒ **Umfassende Befehls-Referenz mit vielen Programmier-Anregungen**

1. Warum Assembler-Programmierung?

Es ist heute oft schwer zu vermitteln, warum man sich im Zeitalter der Ausreife höherer Programmiersprachen ausgerechnet mit Assembler beschäftigen soll, sprechen doch gravierende Vorbehalte eher gegen dessen Einsatz:

- Assembler ist schwierig: Im Prinzip braucht es eine längere Zeit des Einarbeitens. Durch die notwendige 1:1 Übersetzung (1 Befehl in Assemblersprache wird in einen Befehl Maschinensprache übersetzt). Zu recht bezeichnet man deshalb Assembler auch als Maschinensprache, zumindest aber als maschinenorientierte Sprache.

- Die Fehlersuche in Assembler ist mühsam: Man hat nicht die komfortablen Entwicklungsumgebungen der Hochsprachen. Im ungünstigen Fall kann es passieren, dass Assembler-Code nach der Ausführung einfach „abstürzt“. Bedingt durch die höhere Anzahl an Programmanweisungen in Assembler sind die Fehlerquellen natürlich auch umfangreicher.
- Assemblercode ist nicht portabel: Jede Prozessorfamilie hat seine eigene Assemblersprache. Ein Quellcode für den Zilog-Prozessor läuft nicht auf der Intel-Prozessor-Familie. Der „Prozessor-Umsteige-Programmierer“ muss im Prinzip eine neue Programmiersprache erlernen.

Diesen doch schwerwiegend anmutenden Nachteilen stehen natürlich Vorteile gegenüber, die, geschickt genutzt, oben konstatierte Nachteile ausgleichen können. Zu bedenken ist jedoch im Vorfeld, dass es fast unsinnig wäre, wollte man die mittlerweile sehr ausgereiften Hochsprachen zugunsten Assemblerprogrammierung gar nicht mehr nutzen. Folgende unumstößliche Vorteile sind der Assembler-Programmierung inhärent:

- Assemblercode ist der schnellste ausführbare Code. Keine Hochsprache, die die Vorteile des schnellen Codes für sich requiriert kann hier mithalten. Wer also zeitkritische Routinen schreiben will oder muss, kommt um Assembler-Programmierung nicht herum. Code-Optimierung führt unweigerlich zu Assembler! Für jeden Assemblerbefehl kann exakt die notwendige Zeitdauer (bzw. die Anzahl der benötigten Takte) ermittelt oder in Erfahrung gebracht werden.
- Einzelne Komponenten des Prozessors, seien es ALU (Arithmetisch-Logische Einheit), Register oder Coprozessor(en) sind über Assemblercode direkt ansprechbar. Die in Hochsprachen verfügbaren vorgegebenen Funktionen oder Routinen lassen oft eine „elegante“ Lösung vermissen; diese ist oft relativ einfach in Assembler zu realisieren.
- Komprimierte ausführbare Programme: Die mächtigen Programmierwerkzeuge erzeugen, bei genauerem Hinschauen, immer größer werdende ausführbare Programme, was auch in dem Tatbestand begründet liegt, dass nicht nur benötigte Programmbibliotheken den ausführbaren Code aufblähen. Assembler-Programme sind zwar bezüglich des Quellcodes umfangreich, als übersetzte Lademodule sind sie jedoch speicherplatzsparend.
- Verständnis bezüglich des „Innenlebens“ des Rechners: Boshaft formuliert, könnte man konstatieren, dass der Hochsprachen-Programmierer vom „Innenleben“ des Rechners keine große Ahnung zu haben braucht. Nicht zuletzt deshalb wurden die Hochsprachen als „problemorientiert“ geschaffen. Dies ist somit auch meist kein Nachteil. Der reine Anwendungs-Programmierer ist sicherlich froh, sich nicht mit Details des Prozessors auseinandersetzen zu müssen. Andererseits erhöht es sicherlich die Qualifikation eines Programmierers, wenn er Zusammenhänge bezüglich der Hardware in seine Programmier-Aktivitäten miteinbezieht!

Vielleicht liegt der „Königsweg“, wie immer, zwischen den Polen. Das bedeutet in diesem Zusammenhang, dass man Synergieeffekte der Sprachen nutzen sollte, um so letztlich auch deren Nachteile vermeiden lernt. In vielen Fällen wird im Profi-Programmierungsbereich die sog. „Mischprogrammierung“ forciert, die „oberflächenorientierte“ Funktionen eher den Hochsprachen überlässt und die zeitkritischen Routinen in Assembler schreibt.

2. Einordnung der Sprache Assembler in den Kanon der Programmiersprachen

Assembler war in der 2. Computer-Generation, nachdem in Generation 1 Festverdrahtungen lediglich eine Eingabe von Daten über Schalter, die entweder die Stellung **ein** (symbolisiert mit 1) oder **aus** (symbolisiert mit 0) zuließen, eine Weiterentwicklung der dualen bzw. hexadezimalen Kommunikation mit dem Rechner. Sogenannte **Mnemonics**, dem menschlichen Vokabular entnommen, gestatteten das Erfassen von Programmcode, deren Abspeicherung, Ausführung und Definition von Befehlen und das Entfernen des abgearbeiteten Programmes aus dem Internspeicher. Hardwaremäßig war die Programmierung in Assembler verbunden mit der bahnbrechenden Erfindung John von Neumanns, der den Weg für flexible Speicherprogramme legte. Mitte der Fünfziger Jahre des letzten Jahrhunderts wurde Assembler zur ersten Programmiersprache. Der Name ASSEMBLER bedeutet so viel wie „Montage“ (to assemble → montieren). Im Unterschied zu den problemorientierten Sprachen, deren Entwicklung ebenfalls nicht lange auf sich warten ließ (eine frühe Version von FORTRAN wurde 1956 und COBOL wurde 1960 fertiggestellt), handelt es sich bei Assembler um eine sprachliche Umschreibung der Maschinensprache (vgl. dazu die genaue Beschreibung des Programmes BA_OSS1.ASM).

2.1 Unterschied Assembler- / Höhere Programmiersprachen

Aus der Sicht der Philosophie der Programmierung handelt es sich um weitgehende Unterschiede. Bei den Höheren Programmiersprachen haben sich strukturierte und objektorientierte Ansätze herausgebildet, die ein sehr systematisches Vorgehen erfordern. Instrumente zur Vorbereitung der Programmierung sind Struktogramme (bei blockstrukturierenden Sprachen wie C, Pascal oder Modulo). Bei den Objektorientierten Sprachen haben sich Klassen- und Objekt-Diagramme (UML-Diagramme) herausgebildet.

Beim Programmieren in Assembler greift man wieder gerne auf Programmablaufpläne zurück, da Wiederholstrukturen in den meisten Fällen auf Sprung-

befehle zurückgehen. Der Programmieraufwand bei Assembler-Quellcodes ist wesentlich höher (man bedenke, dass der Befehl des Bildschirmlöschens in Hochsprachen aus einem Befehl, in Assembler hingegen aus 5 bis 7 Befehlen besteht). In Assembler gibt es allerdings die Möglichkeit, Befehlsketten zu sog. MAKROS zusammenzufassen bzw. Programm-Routinen für mehrere Programme zur Verfügung zu stellen.

Letztendlich verfügt jede Assemblersprache über lineare Strukturen, Kontroll- und Wiederholstrukturen sowie Unterprogrammaufrufe, so dass der versierte Hochsprachen-Programmierer schnell Zugang zu Assembler findet.

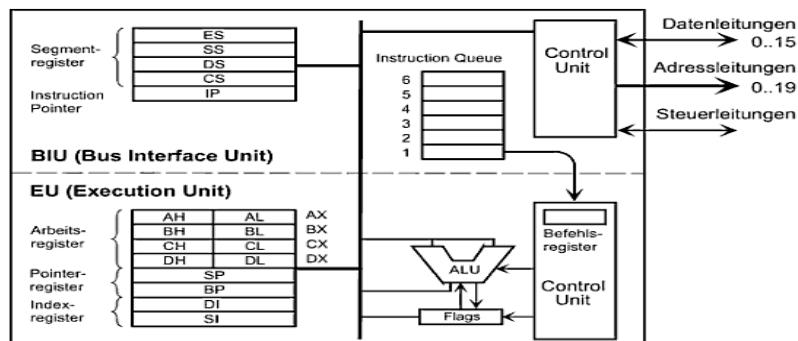
Weiter unten wird außerdem zu klären sein, dass die 16-Bit-Assembler-Programmierung per Textbildschirm stattfindet. D.h., dass die Anwendung von MS-DOS-spezifischen Befehlen, wie z.B. dem Erstellen von Batch-Dateien (mit dem Suffix .BAT) oder der Gebrauch von Befehlen wie TYPE, DIR, MD, CD, REN oder COPY weiterhin Bedeutung zukommt.

2.2 Assemblersprache als Abbild der Maschinensprache

3. Grundlegender Aufbau der Intel-Prozessoren

3.1 8086 als 16-Bit Grundmodell

Der 8086 Prozessor besteht aus 2 grundlegenden Einheiten:



- der BUS INTERFACE Einheit (BIU); diese ist für die Verwaltung des Daten-transportes zuständig. Ihre Hauptaufgabe besteht darin, alle externe Bus-operationen zum Hauptspeicher oder zu den Ein- und Ausgabegeräten zu koor-

dinieren. Intern wird eine Befehlswarteschlange aufgebaut, die ständig abgearbeitet wird. Selbst wenn das Steuerwerk keine Daten über die BIU anfordert, sorgt diese dafür, dass bereits der nächste Befehl die Warteschlange auffüllt. Dadurch wird eine Steigerung der Geschwindigkeit erzielt.

- der EXECUTION UNIT (EU): Sie ist die Verarbeitungseinheit des Prozessors. Die EU greift auf die Befehlswarteschlange, die durch die BIU verwaltet wird, zu. Bevor die Steuereinheit aktiv wird, kommt der neue Befehl in ein Befehlsregister, wo er analysiert wird. Dort wird z.B. festgestellt, ob weitere Operanden zur Befehlsausführung benötigt werden. Diese werden ggf. aus dem Hauptspeicher (Arbeitsspeicher, RAM) nachgeladen. Ist der Befehl komplett, dann wird er an die ALU (Arithmetisch-Logische Einheit) weitergeleitet.

Die Befehlsabarbeitung geschieht im 8086 in einer festgelegten Reihenfolge. Die zeitliche Reihenfolge wird durch die Taktfrequenz bestimmt. Dieser „Puls“ des Rechners wird durch den Taktgenerator erzeugt. Man spricht in diesem Zusammenhang auch vom Maschinenzyklus.

Ein Beispiel soll die Zusammenhänge verdeutlichen. In Assembler könnte eine Anweisung

ADD zahl, 5

lauten, was bedeutet, dass zum aktuellen Inhalt der Variablen ZAHl der Wert 5 hinzuaddiert wird. Grundlage ist hier das sog. Befehlszählregister (IP = Instruction Pointer). Dieser zeigt während der Programmabarbeitung immer auf den nächsten auszuführenden Befehl. Dieser wird zur Analyse in das Befehlsregister transferiert. Eine kleine Schwierigkeit besteht darin, dass die komplette Adresse eines abzuarbeitenden Befehls mit 20 Bit dargestellt wird. Der Befehlszeiger kann jedoch nur 16 Bit adressieren. Also muss zusätzlich die Adresse des Code-Segments ins Befehlsregister geladen werden.

Angenommen, die 16-Bit-Adresse des Befehls lautet **1025** und der Inhalt des CS-Registers (Code-Segment) sei **0985**, dann wird im Befehlsregister die Adresse

A875 zugrunde gelegt. (Die Adresse des CS-Registers wird mit 10H multipliziert und der Offset dazu addiert; merke: hierbei handelt es sich wieder ausnahmslos um HEX-Code!) Die entsprechende Adresse wird über den Adressbus angesprochen; gleichzeitig ergeht ein Impuls zur Datenübertragung an den Steuerbus.

Im **2. Schritt** wird vom Prozessor eine Überprüfung der Bauteile Hauptspeicher und Datenbus vorgenommen. Wenn eine der relevanten Systeme nicht bereit ist, wird vom Prozessor ein „Warteschritt“ („waitstate“ genannt) eingebaut; Erst nach Beendigung eventueller waitstates ergeht ein Impuls zur Datenübertragung an den Steuerbus.

Schritt 3 zeigt die Bereitschaft der Teile Arbeitsspeicher, Datenbus und Prozessor zur Datentransaktion. Die Richtung des Datentransfers (entweder Arbeitsspeicher – Prozessor oder umgekehrt) wird durch den Steuerbus geregelt.

Alle 3 Verarbeitungsschritte bilden einen Operationszyklus. Im äußersten Fall können weitere 2 Schritte hinzukommen: Dies ist bei Rechen- oder Vergleichsoperationen in der ALU notwendig.

Operationszyklen: Holen, Lesen, Ausführen und Schreiben

Holen: Der Prozessor holt im Arbeitsspeicher einen Maschinenbefehl vom Programm in das Befehlsregister.

Lesen: Der Prozessor benötigt zusätzlich einen Speicheroperanden, in unserem Fall die Variable *zahl*.

Ausführen: Der Maschinenbefehl wird durch das Rechenwerk ausgeführt.

Schreiben: Wurde ein Speicheroperand, in unserem Fall *zahl verändert*, so muss dieser wiederum in den Arbeitsspeicher zurückgeschrieben werden.

Da jeder Maschinenbefehl aus einem bis zu 5 Operationszyklen besteht und diese wiederum aus 3 bis 5 Operationsschritten, kommen leicht um die 20 Operations-schritte je Befehl zusammen.

Der Befehl *add zahl,5* benötigt genau 17 Operationsschritte. Bei einer Taktfrequenz von 2 GHz erledigt der Rechner ca. 2 Mrd. Takte pro Sekunde. Also benötigt er 0,5 Nanosekunden zur Ausführung eines Taktes. Zur kompletten Abarbeitung wäre eine Verarbeitungszeit von $17 * 0,5 = 8,5$ nsec notwendig.

3.2 Das „Vererbungskonzept“: Nachfolgeprozessoren

Allen Nachfolgeprozessoren von Intel ist gemeinsam, dass sie alle abwärtskompatibel zum 8086 sind. Der neueste Pentium-Rechner ist in seinem „Innern“ ein erweiterter 8086-Prozessor. Dies erinnert in starken Maße an das Vererbungskonzept der Objektorientierten Programmierung. Zuerst muss die Basisklasse mit all ihren Attributen und Methoden komplett und lückenlos zur Verfügung stehen, erst dann können daraus Klassen abgeleitet werden, die automatisch auf die Methoden (in unserem Falle Befehle des 8086) zugreifen und sie integrieren. Die ableitbaren Klassen beinhalten die Befehle der Nachfolgeprozessoren des 8086.

Demnach ist jeder „neue“ Assembler-Programmierer auf INTEL-Basis angehalten, sich mit dem 8086 programmiertechnisch intensiv auseinander zu

setzen, um dann weiterführende Befehle (z.B. zur MMX-Technologie oder zur 32-Bit-Windows-Programmierung) gewinnbringend nutzen zu können. Zum besseren Verständnis des Zusammenhangs seien die Haupt-Daten und –kriterien des 8086 und seiner Nachfolgeprozessoren in einem Schema dargestellt: (Quelle: Backer, Reiner: a.a.O., S. 52f.)

.... bis zum 80486

Intel-Prozessor	8086	80286	80386 -DX	80486-DX/DX2/DX4
Registergröße	16	16	32	32
Adressbus (Bit)	20	24	32	32
Datenbus (Bit)	16	16	32	32
Adressierbarer Speicher (MB)	1	16	4096	4096
Taktfrequenz (MHZ)	4,7 bis 10	8-16	16-40	20-50/33-66/75-100
Koprozessor	Nein	Nein	Extra Chip notwendig	integriert
Anzahl Transistoren	29000	134000	275000	1,25/2/2,5 Mio
Markteinführung	1978	1982	1985	1989

.... und hier ab Pentium MMX

Intel-Prozessor	Pentium MMX	Pentium II	Pentium III	Pentium IV
Registergröße	32	32	32	32
Adressbus (Bit)	32	36	36	36
Datenbus (Bit)	64	64	64	64
Adressierbarer Speicher (MB)	4 GB	64 GB	64 GB	64 GB
Taktfrequenz (MHZ)	60-200	233-450	450-1330	1300-3200
Koprozessor	integriert	integriert	integriert	integriert

Anzahl Transistoren	3,1-3,3 Mio.	7,5 Mio.	9,5 – 28,1 Mio.	42-55 Mio.
Markteinführung	1993	1997	1999	2000

4. Die Register des 8086 /Nachfolge-Prozessoren

Allgemeine Register

Die allgemeinen Register sind 16 Bit breit, können aber jeweils in zwei 8 Bit breite Register aufgeteilt werden. Das niederwertige Register wird mit dem Buchstaben L, das höherwertige Register mit dem Buchstaben H gekennzeichnet. In Verbindung mit dem Buchstaben A, B, C oder D ergibt sich ein vollständiger Registername. Zum Beispiel wird aus AX (16 Bit breit) das Register AL (8 Bit) und das Register AH (8 Bit). Änderungen an AL oder AH schlagen sich immer auch auf AX nieder.

Bits 15-8	Bits 7-0
AH	AL
AX	

Aufbau

Und hier die einzelnen Register

Register	Bedeutung
AX	Akkumulator Wird häufig für arithmetische Operationen(Division, Multiplikation)verwendet Der höherwertige Teil AH nimmt häufig die Funktionsnummer bei interruptbasierten Operationen auf (s. erstes Programm).
BX	Basis Wird meist bei der Adressierung von Werten im Speicher verwendet
CX	Counter Vielfach in Schleifenstrukturen als 'Laufvariable' eingesetzt
DX	wird ebenfalls für die Adressierung eingesetzt

Allgemeine Register

Einige Befehle oder Routinen arbeiten mit bestimmten Registern. In diesen Fällen ist die Benutzung der Register genau an den Befehl gebunden. Abgesehen von diesen speziellen Aufgaben können diese Register jederzeit verwendet werden, zum Beispiel um Werte im Prozessor statt im Speicher zwischenzulagern.

Segmentregister

Um diese Register zu erklären, muss etwas weiter ausgeholt werden. Wie bereits gesagt wurde, verfügt der 8086 über einen 16 Bit breiten Datenbus. Zusätzlich enthält er auch noch einen Adressbus, den man sich als ein Bündel von Adressleitungen vorstellen kann. Dieses Bündel umfasst 20 Leitungen. Adressierbar sind damit also 2^{20} Bytes (entspricht 1048576 Bytes oder ein Megabyte).

MSDOS kann nur ein Megabyte Speicher ansprechen. Das ist allerdings weniger ein Problem von DOS, sondern liegt eher in der verzahnten Entstehung von DOS und dem 8086. Aus Kompatibilitätsgründen wurde aus dieser Begrenzung später auch kein richtiger Ausweg gesucht.

Um auf eine Adresse im Speicher zugreifen zu können, muss sie in einem Register hinterlegt sein. Die Register des 8086 waren aber nur maximal 16 Bit breit. Mit 16 Bit lassen sich aber nur $2^{16}=65536$ Bytes (64 KB) ansprechen.

Um diesem Dilemma aus dem Weg zu gehen, entschloss man sich, zwei Register für die Adressierung zu verwenden. Damit ließen sich dann immerhin 2^{32} Bytes adressieren. Da der Adressraum des 8086 aber nur ein Megabyte (2^{20}) umfasste, blieben von 32 Bit (4 GB) 12 Bit unbenutzt ($32-20=12$). Aus diesem Grunde entstand bei Intel auf die Idee der Segmentierung.

An welchen Adressen beginnen nun Segmente? Dividieren wir die maximale Größe des Adressraumes durch die maximal mögliche Anzahl an Segmenten, so erhalten wir $2^{20} / 2^{16} = 2^4 = 16$. Ein Segment ist also ein Block mit einer Größe von 16 Bytes. Ein Segment kann demzufolge an jeder Adresse beginnen, die ohne Rest durch 16 teilbar ist. In das Segmentregister muss die Nummer des Segmentes eingetragen werden. Dessen Adresse errechnet sich wie folgt:

*Segmentnummer * 16 Bytes.*

Bis jetzt können wir nur auf Segmente respektive Segmentgrenzen zugreifen. Um den Zugriff auf einzelne Bytes innerhalb der Segmente zu ermöglichen, wird ein zweites Register verwendet. Dieses enthält einen Zeiger auf ein bestimmtes Byte, letztlich auch nur eine Zahl. Um also das 4. Byte im 6. Segment ansprechen zu können, muss ins Segmentregister der Wert 6 und in das zweite Register der Wert 4 eingetragen werden. Eine vollständige Adresse errechnet sich folgendermaßen:

*Segmentnummer * 16 + Offset(Abstand) zum Segmentanfang*

Die logische Darstellung von Segment und Offset sieht so aus:

Segment:Offset,

also zum Beispiel 00006:00004 (üblicherweise erfolgt die Darstellung in hexadezimaler Schreibweise)

Mit einem 16-Bit-Register kann man wesentlich mehr als nur die 16 Bytes bis zur nächsten Segmentgrenze adressieren. Dies bedeutet, dass man mit diesem Zeigerregister über mehrere Segmentgrenzen hinweg operieren kann. Im Umkehrschluss bedeutet das auch,

dass eine lineare Speicheradresse mit verschiedenen logischen Adressen ansprechbar ist. So ist die logische Adresse 00006:00004 gleichbedeutend mit der logischen Adresse 00005:00020 (- es ergibt sich die physikalische Adresse 100).

Und weil mit einem Zeigerregister 64 KB adressierbar sind, ist es tatsächlich möglich, mit einer Segment:Offset-Kombination mehr als ein Megabyte anzusprechen. Dies ergibt sich aus folgender Rechnung:

max. Anzahl Segmente: 65536

Größe eines Segments: 16 Bytes

max. Größe des Offsets: 65535 Bytes

*physikalische Adresse: Segmentnummer * Segmentgröße + Offset*

*max. ansprechbare Adresse: $65536 * 16 + 65535 = 1114111$*

Der Überhang über ein MB wird unter DOS als High Memory Area (HMA) bezeichnet. Dieses Feature ist aber nicht beim 8086 erhältlich, da er nur über 20 Adressleitungen verfügt, aber 21 Leitungen (Bits) für diese Adressierung notwendig sind.

Ein Segmentregister ist immer 16 Bit breit und lässt sich im Gegensatz zu den allgemeinen Register nicht in einen nieder- und einen höherwertigen Teil zergliedern. Es existieren die folgenden Segmentregister

Register	Bedeutung
CS	enthält die Nummer des Segmentes, das den aktuellen Code enthält siehe Informationen zum Register IP
DS	enthält die Nummer des Segmentes, das die Daten enthält Der Offset kann in verschiedenen Registern stehen oder als Konstante übergeben werden
ES	findet vor allem bei den Stringoperationen Verwendung in diesen Fällen steht der Offset im Register DI
SS	enthält die Nummer des Segmentes, das den Stack enthält Offsets stehen in den Register BP oder SP Die Bedeutung des Stack wird später geklärt.
FS	erst ab dem 80386 Dient vor allem als Zusatzsegmentregister, wird hauptsächlich im Protected Mode verwendet.
GS	erst ab dem 80386 Dient vor allem als Zusatzsegmentregister, wird hauptsächlich im Protected Mode verwendet.

Segmentregister

Indexregister

Im Befehlssatz des 8086 gibt es die sogenannten Stringbefehle. Den Begriff String darf man an dieser Stelle aber nicht mit gleichlautenden Datenstrukturen aus Hochsprachen verwechseln. Ein String ist für den Prozessor eine Aneinanderreihung von Daten eines bestimmten Typs (Byte, Word). Dazu sei gesagt, dass der Prozessor Zeichen (**char**) als Bytes behandelt. Der Prozessor sieht lediglich den ASCII-Code (oder eine andere numerische Codierung), die Interpretation als Zeichen muss der Programmierer vornehmen. Aufgrund der prozessorseitigen Interpretation als Zahl ist es sehr einfach, mit Buchstaben zu rechnen.

Die Stringbefehle umfassen das Kopieren aus und Schreiben in einen String, Kopieren eines ganzen Strings oder Teile davon, den Stringvergleich und das Absuchen von Strings.

Die Indexregister übernehmen zusammen mit einem vom jeweiligen Stringbefehl abhängigen Segmentregister die Aufgabe, auf die nächste zu bearbeitende Adresse im String zu zeigen.

Wie die Segmentregister sind die Indexregister 16 Bit breit und nicht teilbar.

Es gibt die folgenden Indexregister

Register	Bedeutung
DI	Destination Index Kann im Grunde frei verwendet werden, dient aber bei den Stringbefehlen in Verbindung mit dem Register ES als genaue Adressangabe.
SI	Source Index; das zum Destinations-Index Gesagte kann auch hier Anwendung finden. Bei Stringanwendungen muss der Quellindex per SI adressiert werden.

Indexregister

Der Stack und die Pointerregister

Stack bedeutet übersetzt soviel wie Stapel, der als **LIFO**-Speicher organisiert ist. **LIFO** steht für **Last In First Out**. Dessen Gegenteil ist der FIFO-Speicher (First In First Out). In der Praxis bedeutet LIFO, dass das letzte Element, das auf den Stapel gelegt wird, auch als erstes wieder vom Stapel entfernt wird.

Im Umfeld eines Prozessors dient der Stack hauptsächlich als Zwischenspeicher für Adressen, wenn es um Unterprogramme geht, dient er auch als Medium, um Parameter zu halten. Ein weiteres wichtiges Anwendungsgebiet ist das Übertragen von Variableninhalten, wenn Assemblermodule in Hochsprachen eingebunden werden.

Exakte Funktion des Stack: Wenn der Prozessor ein Unterprogramm aufruft, dann muss die Adresse des Hauptprogrammes gesichert werden; wenn das Unterprogramm abgearbeitet wird, wird mit im Hauptprozess mit der exakt nächsten Anweisung fortgefahren:

```

Programmbehl1
Programmbehl1
Programmbehl1
Programmbehl1
Prozeduraufruf
Programmbehl5
.
.
.
```

Programmbefehle

Der Prozessor speichert somit die Rücksprungadresse, damit das Programm ordnungsgemäß mit Programmbefehl 5 fortfahren kann.

Bei größeren Programmen bzw. bei extensiven Stack-Operationen bietet sich die Definition eines separaten Stack-Segments an.

Oft wird der Stack aber auch als temporärer Zwischenspeicher definiert. Besondere Bedeutung haben in diesem Zusammenhang die Befehle PUSH und POP.

PUSH legt einen Wert auf die Spitze des Stapelspeichers ab, POP entnimmt dessen Inhalt wiederum. Weiter unten wird zu erkennen sein, dass dadurch auch ein Transfer zwischen Registern stattfinden kann:

PUSH DS → legt die Adresse des Datensegmentes auf den Stapel

POP ES → entnimmt die gleiche Adresse und weist sie dem Extra-Segment-Register zu.

Üblicherweise sollte jedes Programm über einen Stack verfügen. Sollte dies nicht der Fall sein und es wird trotzdem Platz auf dem Stack benötigt, dann werden die Stack-relevanten Inhalte aufrufenden Programm zwischengespeichert; um der Gefahr des Stack-Overflow zu begegnen ist es ratsam ein Stack-Segment in ausreichender Größe zu definieren.

Als Besonderheit des Stacks ist zu beachten, dass die Adresse der aktuellen Stackspitze von oben nach unten wächst, d. h. je mehr Elemente auf den Stack gelegt werden, desto niedriger ist die Adresse der Stackspitze.

Nach Definition des Stack-Segments mit der Bestimmung der Stack-Größe, wird aber erst nach dem Programmstart dessen genaue Position festgelegt. Gerade rekursive Programmaufrufe überlasten oft das gesamte Stacksegment. Daher ist es nicht immer einfach, die exakte Größe des Stacks zu antizipieren.

Register des Stacks:

Register	Bedeutung
SP	Stack Pointer Dieses Register zeigt auf die aktuelle Stack-Spitze
BP	Base Pointer Dieses Register zeigt auf die für den aktuellen Kontext gültige Stackbasis. Meist verwenden Unterprogramme einen eigenen sogenannten Stackrahmen und mit Einbeziehung dieses Registers lassen sich Adressen von übergebenen Parametern errechnen.

Pointerregister

Weitere Register

Das Flagregister

Dieses Register wird auch Prozessorzustandsregister genannt. Es ist 16 Bit breit und jedes einzelne Bit spiegelt einen bestimmten Zustand nach einer Operation des Prozessors wider. Meistens sollen abhängig von diesem Zustand bestimmte Aktionen durchgeführt werden.

Eine Verzweigung in einem Assemblerprogramm ist nur über Sprünge realisierbar. Im Befehlssatz des 8086 befinden sich die sogenannten bedingten Sprungbefehle, die die Flags auswerten und entsprechend zu anderen Programmstellen verzweigen.

Beim 8086 sieht es folgendermaßen aus ('_' bedeutet: reserviertes oder noch nicht belegtes Bit).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
–	–	–	–	Overflow	Direction	Interrupt Enable	Trap/Single Step	Sign	Zero	–	Auxiliary	–	Parity	–	Carry

Das Flagregister des 8086

Man spricht von einem gesetzten Flag (und Bit allgemein), wenn es den Wert 1 hat. Im umgekehrten Fall ist das Flag (Bit) nicht gesetzt bzw. ist es gelöscht.

Um einen Rückgabewert innerhalb von MS-DOS verstehen zu können, muss man die Wertigkeit der nicht veränderbaren Bits miteinbeziehen:

- ⇒ das 1. Bit speichert immer den Wert 1
- ⇒ das 3. Bit speichert immer den Wert 0
- ⇒ das 5. Bit speichert immer den Wert 0
- ⇒ das 12. Bit speichert immer den Wert 1
- ⇒ das 13. Bit speichert immer den Wert 1

Außerdem ist der Default-Inhalt des 9. Bits (Interrupt-Enable-Flag) 1 (anstatt 0).

Somit wäre der Default-Wert des Flag-Registers direkt nach dem Programmstart 3202HEX: 0011 0010 0000 0010

Carry-Flag (Bit Nr. 0)

Dieses Flag wird sehr vielfältig verwendet. An erster Stelle signalisiert dieses Flag einen Überlauf bei vorzeichenlosen Zahlen, verursacht zum Beispiel durch die Addition von 100 zu 65530 in einem Word-Operanden (das Ergebnis ist nicht mehr in einem Word darstellbar).

Eine Reihe mathematischer Befehle beziehen dieses Flag in ihre Funktionsweise ein, genau wie diverse Bitschiebebefehle (z.B. ROR) sowie bedingte Sprungbefehle, doch dazu mehr bei den Erklärungen zu den Befehlen. Ab dem 80386 werden Befehle zur Verfügung gestellt, mit denen das Carry-Flag explizit gesetzt werden kann.

Parity-Flag(Bit Nr. 2)

Dieses Flag zeigt an, ob die Anzahl der gesetzten Bits in den unteren acht Bits eines Resultats gerade (1) oder ungerade (0) ist.

Auxiliary-Flag (Bit Nr. 4)

Dieses auch Hilfsüberlauf-Flag genannte Flag hat seine Bedeutung bei der sogenannten BCD-Arithmetik. Mit dieser Form der Arithmetik wird verhältnismäßig selten gearbeitet.

Zero-Flag(Bit Nr. 6)

Der Befehlssatz des 80x86 enthält viele Befehle, die das Zero-Flag setzen, wenn das Resultat ihrer Ausführung den Wert Null ergibt.

Dieses Flag wird häufig bei Vergleichen (die auch nur Berechnungen darstellen) genutzt sowie beim Test, ob einzelne Bits in einem Operanden gesetzt sind oder nicht.

Beachten Sie bitte, dass dieses Flag gesetzt ist, wenn das Ergebnis einer Operation Null ergibt. Lassen Sie sich nicht von den Werten Null und Eins für einzelne Bits verwirren.

Sign-Flag (Bit Nr. 7)

Dieses Flag wird gesetzt, wenn das Ergebnis einer Berechnung das höchstwertige Bit im Zieloperanden setzt. Bei vorzeichenbehafteten Zahlen ist dies ein Zeichen für einen negativen Wert.

Trap/Single Step-Flag (auch: **Trace-Flag**)(Bit Nr. 8)

Dieses Flag bietet die Möglichkeit, nach jedem Befehl den Programmablauf zu unterbrechen und, wofür dieses Flag meistens eingesetzt wird, einen Debugger mit der Registerauswertung zu beauftragen.

Ist dieses Flag gesetzt, dann kann mittels Debugger der Code im sogenannten Einzelschrittmodus geprüft werden.

Es gibt keinen direkten Befehl, mit dem dieses Flag beeinflusst werden könnte, vielmehr muß eine Befehlskombination angewendet werden, die das Flagregister auf den Stack pusht, von dort in ein Register poppt, dann muß mittels eines geeigneten Befehls das Trace-Flag geändert werden, dann auf den Stack pushen und von dort ins Flagregister poppen.

Interrupt Enable-Flag(Bit Nr. 9)

Der 80x86 kann auf externe Unterbrechungen, sogenannte Interrupts reagieren. Einige Programme (z.B. Interrupthandler selbst oder TSRs) dürfen nicht unterbrochen werden. Um das zu erreichen, muss dieses Flag gelöscht sein.

Direction-Flag(Bit Nr.10)

Dieses Flag wird von den Stringbefehlen der CPU verwendet.

Wenn es gesetzt ist, dann wird von der Startadresse ausgehend automatisch auf die

nächstniedrigere Adresse zugegriffen, ist es gelöscht, dann erfolgt der Zugriff in umgekehrter Form. Üblicherweise ist bei Stringoperationen das Direction-Flag gelöscht.

Overflow-Flag(Bit Nr. B)

Dieses Flag wird gesetzt, wenn bei einer mathematischen Operation das Ergebnis nicht mehr in den vorzeichenbehafteten Zieloperanden passt.

Das Vorzeichen wird durch das höchstwertige Bit repräsentiert, in einem Byte also Bit 7. Ist dieses Bit gesetzt, dann ist die Zahl negativ.

Das **Flag-Register** kann mit dem Befehl *PUSHF* (push flag) gespeichert und mit *POPF* (pop flag) wiederhergestellt werden. (Vgl. insbesondere S.

IP - Instruction Pointer

Dies ist eines der wichtigsten Register des Prozessors.

Es enthält einen Zeiger auf die Adresse im Codesegment, an der der nächste auszuführende Befehl steht (CS:IP).

Der Programmierer kann dieses Register nicht direkt beeinflussen. Eine Einflussnahme ist nur über die Sprungbefehle oder den Aufruf eines Unterprogramms oder einem Rücksprung aus einem solchen möglich.

II. 16-Bit- Assembler-Programmierung

4. Werkzeuge zur Assembler-Programmierung

- a. Editor
- b. Assembler
- c. Linker
- d. Debugger

DEBUG ist ein Testhilfeprogramm, welches direkt Assemblercode verarbeitet und z.B. im Einzelschrittmodus die Veränderung von Registerinhalten und Flags zeigt sowie RAM-Speicherinhalte als sog. Dumps darstellt. Die DEBUG-Befehle werden nach dem Promptzeichen (" ") als Einzelbuchstaben eingegeben. Die wichtigsten sind:

- G (Go →) startet das Assemblerprogramm und führt es komplett aus. -G 120 startet das Programm und fährt ab Speicherstelle 120 (hex) den Einzelschrittmodus.
- T (Trace →) Einzelschrittmodus; ein Befehl wird abgearbeitet und zeigt danach sämtliche Register an.
- P (Procedure →) wie -T mit dem Unterschied, dass Prozedur-, Schleifen und Interrupt-Routinen ausgeführt, nicht aber deren einzelne Anweisungen ausgeführt werden.
- D (Dump →) Speicherauszug; so zeigt -D 200 den Speicherblock ab der Speicherstelle hexadezimal mit den dazugehörigen Zeichen.
- R (Register →) zeigt Registerinhalte. Über -Rax kann z.B. der aktuelle Inhalt des Registers AX gezeigt werden. Im 2. Schritt kann diesem ein anderer Wert zugewiesen werden.
- A (Assemble →) erfasst Assemblercode. Per -A 100 kann ab der Speicherstelle 100 (hex) des aktuellen Codesegments ein Quellcode abgelegt werden.
- U (Unassemble →) zeigt den Assemblercode ab einer bestimmten Speicherstelle. -U100 zeigt die Anweisungen ab Speicherstelle 100.
- Q (Quit →) verlässt den Debugger.

→ Eine einfache Problemstellung mit dem Debugger bearbeiten:

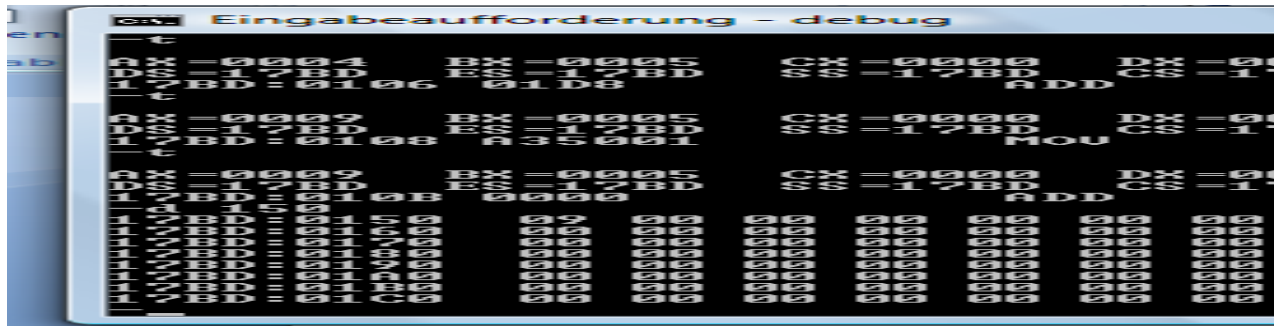
Ab der Speicherstelle 100 hex des aktuellen Code-Segments soll folgendes einfaches Programm im RAM abgelegt werden:

1. Laden des AX-Registers mit dem Wert 4
2. Laden des BX-Registers mit dem Wert 5
3. Addieren beider Register; das Ergebnis soll in AX erscheinen
4. Transfer des Ergebnisses in Speicherstelle 150 des Code-Segments

Programmaufruf: DEBUG

Dazu dienen folgende Assemblerbefehle:

```
-a 100  
  
mov ax,4  
  
mov bx,5  
  
add ax,bx  
  
mov [150],ax
```

Obiges Hardcopy zeigt einen Auszug der Ausführung des Programmes per Trace-Befehl, wobei der jeweils nächste Befehl unter den Registern angeordnet ist. Nach Abarbeitung des letzten Übertragungsbefehls wird per -D 150 die Speicheransicht aufgerufen, die an der Stelle 150 das Additionsergebnis des Programmes zeigt.

Ü1: Erfassen Sie obige Code-Sequenz ab Speicherstelle 100, lassen Sie die Codierung im Speicher zeigen, arbeiten Sie die Befehle im Einzelschrittmodus ab und Zeigen Sie den RAM-Bereich ab Speicherstelle 150 (hex).

Ü2: Schreiben Sie folgende kleine Codierung per DEBUG:

Im AX-Register ist der Inhalt 41 einzutragen.

Der Inhalt von AX ist nach Speicherstelle 160 zu transferieren.

Dann ist AX mit 42 zu belegen. Der Inhalt ist nach Stelle 161 zu bringen.

Als letztes ist BX mit 43 zu belegen. Dieser Inhalt ist nach Speicherstelle 162 zu transportieren.

Speichern Sie die entsprechenden Befehle ab Speicherstelle 100 (hex) ab und stellen Sie fest, dass die Programmzeilen im RAM stehen.

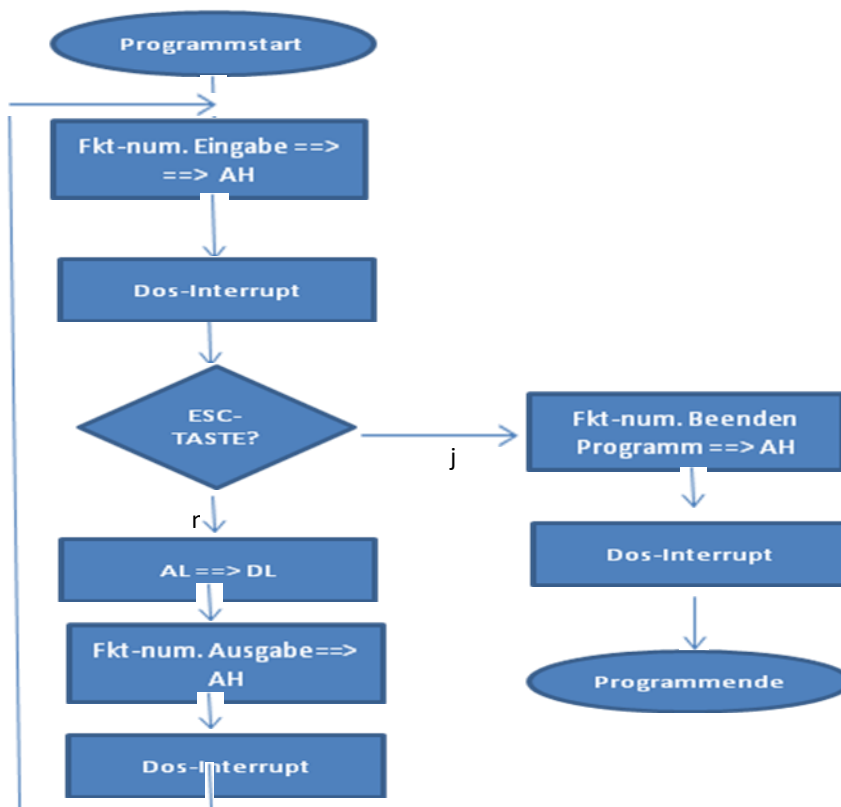
Führen Sie das Programm im Einzelschrittmodus aus und schauen Sie den Inhalt des Speicherbereichs mit dem Zeichenblock ab Speicherstelle 160 an!

5. Vom Quellcode bis zum ausführbaren Programm

Aus Gründen des Handlings werden hier bereits Sprachelemente verwandt, die später näher erläutert werden; einfach um das Programm auch auf Eingabeaufforderungsebene oder als Klick-Icon zu verwenden.

Eine denkbar einfache Problemstellung soll die gezeigten verschiedenen Stufen durchlaufen: Es sollen so lange einzelne Zeichen über Tastatur eingegeben und am Bildschirm gezeigt werden, bis die ESC-Taste betätigt wird. Danach soll das Programm normal beendet werden (d.h. wir landen wieder auf der DOS-Ebene):

- Grobstrukturierung (Programmablaufplan)



- Erfassen des Quellcodes

Benötigt wird ein einfacher Editor, es kann aber auch jedes Textverarbeitungsprogramm verwendet werden.

- Es ist darauf zu achten, dass der Quellcode mit dem Datei-Suffix .ASM abgespeichert wird.
- Das Semikolon im Quellcode dient zur Einleitung eines Kommentars.
- Außerdem werden häufig Sprungziele (Labels) gesetzt, deren Bezeichnung von einem Doppelpunkt gefolgt sein muss.

Unser Einstiegsprogramm sieht folgendermaßen aus:

```

;Programmname: Oss1.ASM *****
;*****
org 0x100                ; Beginne Programm bei Adresse 100H
bits 16                  ; DOS läuft im real-mode -> 16bit
  
```

```

; SMALL beinhaltet hier nur Code-Segment

SECTION .CODE

; M1: ; Anfangs-Label

M1:      MOV AH,08H      ; liest Zeichen von Tastatur ohne Bildschirmecho

          INT 21H        ; allgemeiner Interrupt

          CMP AL,1BH     ; vergleiche Inhalt von AL mit ESC-Taste

          JE ENDE        ; wenn gleich, dann springe zu ENDE-LABEL

          mov dl,al      ; bringe Eingabe-Zeichen ins rechte Daten-Byte

          mov ah,2       ; Funktionsnummer für Bildschirmausgabe

          int 21H        ; allg. Interrupt

          JMP M1         ; unbedingter Sprung zum M1-Label

ENDE:    ; ENDE-Label

          MOV AH,4CH     ; Funktionsnummer „zurück zu DOS“

          Int 21H        ; allg. Interrupt

```

-

- `OSS1.ASM` -

- Assemblieren des Quellcodes (einschließlich .LST-Datei)

Verwendung findet hier der Assembler NASM (Netzbasierter Assembler). Dieser liegt als NASM.EXE vor. Im Prinzip können Assemblier- und Linkvorgang zusammengefasst werden. Im aktuellen Verzeichnis kann dann der Dateiname als Parameter übergeben werden. Dazu wurde eine Batch-Datei entworfen, die über %1 den Dateinamen erwartet. NASM erzeugt unter Angabe des Outputparameters -o eine gleichnamige ausführbare Datei und über -l eine List-Datei, die neben dem Quellcode auch Speicheradressen und den intern erzeugte HEX-Darstellung des Quellcodes zeigt

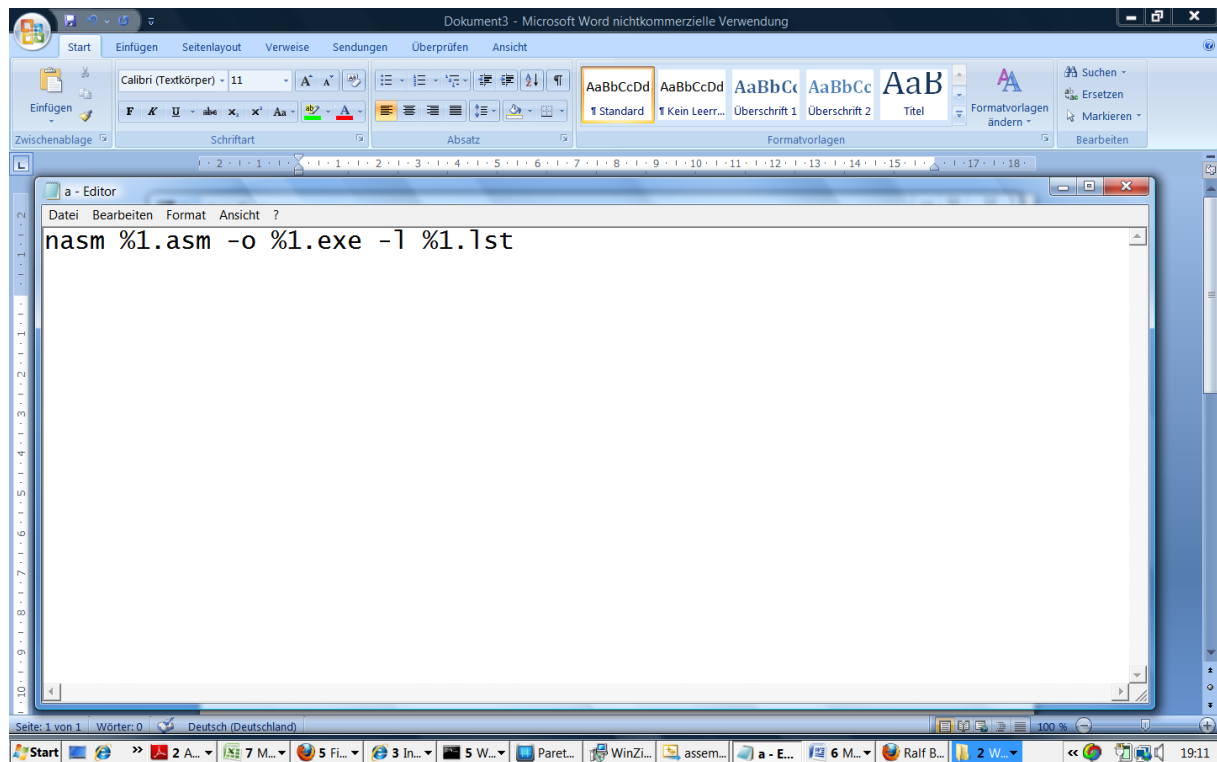
Unten stehendes Eingabe-Fenster zeigt im oberen Teil zunächst den Inhalt der Batch-Datei **a.bat**.

In der 3. Zeile wird hinter dem DOS-Promptzeichen der Batch-Prozess gestartet:

- `a` startet die Batch-Datei
- `OSS1` wird als Dateiname übergeben.

Demnach werden bei Fehlerfreiheit zusätzlich zur bereits abgespeicherten Quellcode-Datei die ausführbare Datei (Lademodul) **oss1.EXE** und die LIST-Datei **oss1.LST** erzeugt.

oss1.EXE ist dadurch auf DOS-Ebene per Eingabeaufforderung ausführbar oder kann gemäß obiger Durchführungsweise per DEBUG ausgetestet werden.



Assembler- und LINK-Vorgang für oss

1

Ein weiterer Screenshot zeigt zunächst den Inhalt des aktuellen Verzeichnisses, wobei die unterschiedliche Größe der erzeugten Dateien ersichtlich wird: Während der Quellcode fast ein Kilobyte an Speicherplatz benötigt und der Umfang der LST-Datei sogar weit darüber hinaus geht, belegt das Lademodul oss1.EXE sparsame 21 Byte!

```

Eingabeaufforderung - debug oss1.exe

H:\ASSEMBLE\NASM\sub>dir oss1.*
Volume in Laufwerk H: hat keine Bezeichnung.
Volumeseriennummer: C00F-2667

Verzeichnis von H:\ASSEMBLE\NASM\sub

14.05.2011  18:55           931 oss1.asm
14.05.2011  18:56             21 oss1.exe
14.05.2011  18:56        1.571 oss1.lst
               3 Datei(en),       2.523 Bytes
               0 Verzeichnis(se), 12.465.668.096 Bytes frei

H:\ASSEMBLE\NASM\sub>debug oss1.exe
~p
AX=0800 BX=FFFF CX=FE15 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
DS=17BD ES=17BD SS=17BD CS=17BD IP=0102  NU UP EI PL NZ NA PO NC
17BD:0102 CD21      INT     21
~

```

Listen der erzeugten Dateien und Aufruf des Debuggers zum Austestvorgang für oss1

Unten stehende Datei oss1.LST gibt einen genauen Überblick über belegte Adressen im aktuellen Code-Segment, den dort im Speicher abgelegten ausführbaren Code in hexadezimaler Entsprechung. Durch Subtraktion zweier aufeinander gelisteten Adressen kann leicht der Speicherbedarf der entsprechenden Assembler-Anweisung abgelesen werden. So belegt z.B. der Assemblerbefehl **MOV AH,08H** in Zeile 6 genau 2 Byte an Speicherplatz (Adresse 00000002 minus 00000000).

```

;*****
;
;Prog.Name: oss1.LST

; Adressen ; Maschinencode

1                org 0x100                ; beginne bei Speicherstelle 100HEX

3                Bits 16                  ; 16-Bit-Code

4                SECTION .CODE

5                M1:                      ;Anfangs-Label

6 00000000 B408                MOV AH,08H    ; liest Zeichen von Tastatur ohne Bildschirmecho
7 00000002 CD21                INT 21H        ; allgemeiner Interrupt
8 00000004 3C1B                CMP AL,1BH    ; vergleiche Inhalt von AL mit ESC-Taste
9 00000006 7409                JE ENDE        ; wenn gleich, dann springe zu ENDE-LABEL
10 00000008 88C2               mov dl,al    ; bringe Eingabe-Zeichen ins rechte Daten-Byte
11 0000000A B402               mov ah,2    ; Funktionsnummer für Bildschirmausgabe
12 0000000C CD21               int 21H        ; allg. Interrupt
13 0000000E E9EFFF            JMP M1        ; unbedingter Sprung zum M1-Label

14                ENDE:                ; ENDE-Label

15 00000011 B44C               MOV AH,4CH    ; Funktionsnummer „zurück zu DOS“
16 00000013 CD21               Int 21H        ; allg. Interrupt

```

MOV wird durch Ziffernkombination B4 und die Funktionsnummer **08** mit führender Null abgelegt. Codetechnisch wird für jede HEX-Ziffer der 4-Bit-Code BCD (Binary Code Decimal) verwandt (B → 1011 4 → 0100 0 → 0000 8 → 1000).

Im übersetzten Objekt-Code wird somit die Assembler-Anweisung

MOV AH,O8H in den Objektcode

1011	0100	0000	1000
------	------	------	------

überführt.

Klar erkennbar ist z.B. auch, dass die Labels (Sprungadressen im Programm) keinen Maschinencode erzeugen:

- **LINKEN**

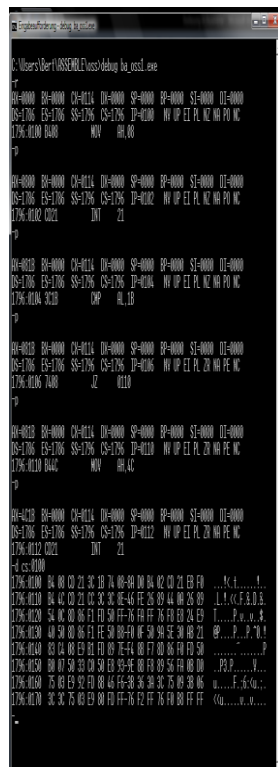
In unseren bisherigen Aktivitäten wurden keine Object-Files erzeugt. Diese Zusammenhänge werden wir an anderer Stelle nachholen, wenn es darum geht mehrere Quellcodes getrennt zu assemblieren oder, per Mischprogrammierung, zu compilieren und/oder assemblieren. Im wesentlichen ist die Aufgabe eines LINKERS nicht nur das (Zusammen-) Binden von Objektmodulen, sondern auch das Ersetzen von symbolischen Adressen (z.B. Labels) in konkrete Speicheradressen.

- **Ausführen der EXE-Datei**

Die assemblierte und gelinkte Datei oss1.EXE wird als ausführbare Datei ins gewünschte Verzeichnis gespeichert. Damit ist sie durch Eingabe des reinen Dateinamens (ohne Suffix) auf der Eingabeebene (oder im Windows-Startmenü unter „Ausführen“) zu starten. Außerdem kann sie durch Doppel-Mausklick aktiviert werden.

Ü3: *Vollziehen Sie obigen Prozess des Editierens, Assemblierens und Erzeugens einer ausführbaren Date mit den*

Starten Sie **DEBUG** (Parameter ist der Name der erzeugten EXE-Datei) und starten Sie den Programmtest per **-P** (Procedure)- Anweisung. Nach jedem Interrupt 21 ist über Tastatur ein Zeichen einzugeben. Durch Betätigen der **ESC**-Taste wird die Eingabe der Zeichen beendet.



6. Verschiedene Segmente

a. Zuweisung von Segmentadressen

Segmente sind auf DOS-Ebene Speicherbereiche von $2^{16} = 64$ KB (oder 65536 Byte). Bei der Benutzung des NASM-Assemblers werden diese Sections genannt. Man unterscheidet bei 16-Bit Systemen die sections CODE, DATEN, EXTRA und STAPEL. Jeder section kann ein Segment-Register zugeordnet werden (CS, DS, ES und SS).

Segmentadressen werden in der Regel vom System selbst zugewiesen. Beim CS-Register ist der Wert nicht beeinflussbar. Über die Anweisung ORG kann bezüglich der Offset-Adresse ein Anfangspunkt festgelegt werden. Durch

ORG 100H wird der zu erstellende Quellcode ab Speicherstelle 100H (=256D) im RAM-Speicher abgelegt. Auch beim DS (=Datensegment-Register) und ES (Extrasegment-Register) kann die relative Anfangsadresse (gezählt ab Offset 0) über den ORG-Befehl zugewiesen werden. Im einfachsten Modell stehen Code und Daten in einem Segment.

Bei String-Operationen (siehe unten) befindet sich die String-Quelle standardmäßig im Datensegment und das String-Ziel im Extra-Segment.

Es ist sinnvoll bei größeren Projekten ein Stapelsegment zu deklarieren.

Um bei kleinen Projekten dafür zu sorgen, dass das Code- und Datensegment im gleichen Speichersegment stehen, wird man zu Beginn des Programmes folgende Anweisungen definieren:

```
SECTION .CODE
```

```
MOV ax, CS
```

```
MOV DS, ax
```

Es besteht die Notwendigkeit diese Zuweisung über ein allgemeines 16-Bit-Register vorzunehmen, da die Zuweisung in folgender Form nicht zulässig ist:

~~MOV DS, CS~~

b. Variablenadressen und -zuweisungen

Soll z.B. ein beliebiger Wert im BX-Register einer Speichervariablen des Datensegments zugewiesen werden, so sollte die Speichervariable einen 16-Bit-Wert speichern können; die Direktive **dw** (define word) ist in diesem Fall

der Variablenbezeichnung anzuhängen. Die Speichervariable soll zunächst mit 0 initialisiert werden. Das Datensegment beinhalte lediglich diese einzige Variable:

SECTION .DATA

Variable dw 0 ; für 16-Bit-Variablen

Möchte man Variablen mit einzelnen Bytes definieren, dann sieht dies wie folgt aus: **db** (define byte) deklariert eine Byte-Variable

Variable db 0 ; für 8-Bit-Variablen

Im folgenden soll die Variable mit dem Wert 15A3H geladen werden. Durch die Verwendung eines eckigen Klammerpaares kann man direkt den Variablen-Inhalt verändern: (Zeile 8 unten stehenden Listings)

MOV word[variable], bx || *entspricht dem Inhalt einer Referenz in der Hochsprachenprogrammierung Die Angabe „word“ vor der Variablenbezeichnung ist obligatorisch.*

Damit ergibt sich das komplette Programm:

```

*****
;
;
; * Verwendung von Variablen *
;
; * Prog.Name: oss2.ASM *
;
*****
;
ORG 100H

Bits 16

SECTION .CODE

Start:                ; Programm-Anfangs-Label

    MOV ax,CS          ; Initialisiere Datensegment per

    MOV DS,ax          ; Ringtausch (beide Teile in der gleichen Section)

    MOV bx, 15A3H       ; lade bx mit beliebigem Hexwert

    MOV word[variable], bx ; Inhalt von bx der Variablen variable zuweisen

    MOV AH, 4CH         ; Programmende

```

```

int 21H

SECTION .Data
variable dw 0 ; Definition einer WORT-Variablen

```

Per Speicherdump ist zu sehen, dass die niederwertigen 8 Bits auch im Speicher die niederwertigen Adressen zugewiesen bekommen. Entsprechendes gilt für die höherwertigen Bits.

Ü4: *Debuggen Sie obigen Quellcode und achten Sie auf das Abspeichern und den Transfer der Variableninhalte.*

Ist man an der effektiven Adresse einer Speicher-Variablen interessiert, so lässt sich dies über die Referenz der Variablen erreichen, die den Abstand der Speicheradresse der Variablen vom Beginn der Data-Section aus beinhaltet.

```

*****
;
;* Verwendung von Variablen - Offset nach CX      *
;
;* Prog.Name: oss3.ASM                            *
;
*****
;

ORG 100H

Bits 16

SECTION .CODE

Start: ; Programm-Anfangs-Label

        MOV ax,CS ; Initialisiere Datensegment per
        MOV DS,ax
        MOV cx, variable ; lade Offset der Variablen-Adresse

```

```

MOV AH, 4CH          ; Programmende
int 21H

SECTION .Data
variable dw 32ABH    ; Definition einer WORT-Variablen

```

16-Bit Variable in Byte-Variablen auflösen:

Möchte man die 8-Bit-Teile der Wort-Variablen getrennt ansprechen, so kann dies über Angabe eines Distanz-Ausdrucks geschehen:

Wir wollen bewirken, dass obiger 16-Bit-Inhalt in zwei 8-Bit-Variablen gespeichert wird:

```

;*****
;
;* Verwendung von Variablen                                     *
;
;* Prog.Name: oss4.ASM                                         *
;*****
;
ORG 100H
Bits 16
SECTION .CODE

MOV ax,CS          ; Initialisiere Datensegment per
MOV DS,ax          ; Ringtausch
MOV cx, variable   ; lade Offset der Variablen-Adresse
MOV BI,byte[variable] ; lade niedrigwertigen 8-Bit-Inhalt
MOV BH,byte[variable+1] ; lade höherwertigen 8-Bit-Inhalt
mov [niedrig],bl    ; kopiere Inhalt in 8-Bit-Var.
mov [hoch],bh       ; kopiere Inhalt in 8-Bit-Var.

MOV AH, 4CH        ; Programmende
int 21H

```

```
SECTION .Data
```

```
variable dw 32ABH ; Definition einer WORT-Variablen
```

```
niedrig db 0
```

```
hoch db 0
```

Der 8-Bit-Inhalt des (niedrigerwertigen) Teils wird zunächst in nach bl und der höherwertige nach bh transferiert. Dazu muss in Zeile 5 der Code-Section einfach die Adresse von variable um einen (Byte-)wert erhöht werden.

Durch Debugging kann man nachvollziehen, dass die jeweils „richtigen“ Inhalte den Byte-Variablen zugeordnet wurden.

```

;*****
;
;* Verwendung von Variablen *
;* Prog.Name: oss5.ASM *
;*****
;

ORG 100H
Bits 16
SECTION .CODE
Start: ; Programm-Anfangs-Label

    MOV ax,CS ; Initialisiere Datensegment per
    MOV DS,ax

    mov bl, 'A' ; Buchstabe A nach bl
    mov [array1],bl ; speichere an Adresse array1
    inc bl ; erhöhe BL (Buchstabe B)
    mov [array1+1],bl ; speichere an Adresse array1+1

    inc bl ; erhöhe BL (Buchstabe C)
    mov [array1+2],bl ; speichere an Adresse array1+2

    inc bl

```

```
mov [array1+3],bl
MOV AH, 4CH          ; Programmende
int 21H

SECTION .Data
array1: Times 5 db 0 ; Byte-Array mit 0 initialisiert
```

Will man Bereiche (z.B. Arrays oder Strings) im Datensegment deklarieren, so spielt der **Times** – Operator eine wichtige Rolle. Hierdurch sind mehrere Inhalte des gleichen Variablentyps (Byte- oder Word- oder Doppelword-Variablen) darstellbar:

Ein BYTE-Array, der 5 8-Bit Zahlen speichern kann, wird wie folgt deklariert:

```
array1: Times 5 db 0
```

5 bedeutet hier die Anzahl der Speicherstellen im Array.

Ü5: Schreiben Sie ein Assemblerprogramm, das obiges Programm Oss5.ASM erweitert:

Nach dem Einspeichern der 4 Byte-Inhalte sollen diese am Bildschirm gezeigt werden.

Ü6: Schreiben Sie ein Assemblerprogramm, das in der Data-Section 3 Wort-Variablen deklariert (Inhalt ist beliebig!); definieren Sie darüber hinaus ein Wort-Adressarray mit 3 Einträgen, der die Offset-Adressen der 3 Variablen speichert. Transferieren Sie am Ende den mittleren Eintrag des Array in der richtigen Reihenfolge (erst highbyte, dann lowbyte) nach CX.

Besonderheit: Beim Initialisieren oder Zuweisen von hexadezimalen Inhalten darf kein Buchstabe (A..F) auf der höchsten Halbbyte-Stelle stehen. Der Assembler bringt dann eine Fehlermeldung, weil bestimmte interne Befehle mit HEX-Ziffern beginnen.

Abhilfe: Man definiert links neben der ersten (Buchstaben-)hexziffer eine 0. Will man z.B. die Variable var1 mit einem Anfangsinhalt FF00H versehen, so ist es not-wendig:

VAR1 DW 0FF00H

zu schreiben. Gleiches gilt beim Zuweisen einer absoluten Zahl ins Register:

MOV AX, 0FF00H.

Ü7: Lösen Sie folgendes Programmierproblem: Definieren Sie zwei Wortvariablen im Datensegment. Weisen Sie beiden hexadezimale Inhalte zu. Tauschen Sie deren Inhalte unter Benutzung von 8-Bit-Teilregistern.

7. Einfache Lineare Programmoperationen

Neue Befehle:

EQU (equate) → Zuweisung von Zahlenkonstanten

(Pseudobefehl)

MUL (multiply) → multipliziert einen Registerinhalt oder eine Konstante mit dem Inhalt von AX

DIV (divide) → dividiert den Inhalt von AX durch eine Konstante oder einen Registerinhalt: Ergebnis wird bei 16-Bit-Operationen in AX:DX gespeichert.

a. Verwendung von Konstanten

Mit der EQU – Anweisung (to equate = gleich setzen) können außerhalb der definierten Segmente Konstanten definiert werden. Dies reduziert den Abstraktionsgrad. Und erhöht die Lesbarkeit des Programmes. EQU bezieht sich bei NASM im Gegensatz zu MASM nur auf Zahlenkonstanten.

```

,*****
;
;* Die EQU - Pseudoansweisung ==> Durchführung einer einfachen *
;* Prog.Name: oss6.ASM          Divisions-Rechnung          *
,*****
rech_betrag EQU 12          ; Konstanten mit Entsprechungen
mwst EQU 19                 ; funktioniert mit numerischen Inhalten
teiler EQU 100
ORG 100H
Bits 16
section .CODE
    mov BX, mwst             ; Weise die Konstante mwst zu
    mov ax, rech_betrag      ; lade AX mit Konstanten rech_betrag
    mul BX                   ; multipliziere AX mit BX, Ergebnis nach AX
    mov cx, teiler           ; Konstante teiler wird cx zugewiesen
    div cx                   ; Ergebnis (Vorkommastelle) steht in
                             ; AX, Nachkommastelle in DX
    ;Beende
    mov ah, 4cH              ; Zuweisung des Beendigungscode Mov ah,4cH
    int 21H                  ; Konstante für Interrupt 21H

```

Ü8: Einfache Lohnberechnung

(Vorbemerkung: Definieren Sie alle Inhalte hexadezimal, dann können die jeweiligen Ergebnisse in Registern und Variable quasi als Dezimalwerte interpretiert werden!)

Verwenden Sie als Konstanten:

tariflohn, die den Wert 2000 speichert;

erhoehung: erhält den Inhalt 3

Definieren Sie eine 16-Bit-Variable: **personalkosten**, welche den Wert des erhöhten Tariflohnes zugerechnet bekommt.

b. Ein- und Ausgabeoperationen

- Einfache Zeichenausgaben

Die dazu gehörigen Beispiele wurden bereits zu Anfang des Skriptums in Verbindung mit DEBUG-Übungen und Anfangs-Betrachtungen getätigt, so dass an dieser Stelle darauf verzichtet werden kann!

- Ausgabe kompletter Register-Inhalte

Oft möchte man gesamte 8- oder 16-Bit-Register-Inhalte am Bildschirm ausgeben. Dies erscheint auf den ersten Blick einfacher zu sein als es wirklich ist. Im folgenden Beispiel sei ein konkreter Inhalt des AX-Registers 35 | 78 , wobei 35 der höherwertige und 78 der niedrigwertige 8-Bit-Anteil des Registers darstellt. Die 4 Ziffern 3,5,7 und 8 sollen nacheinander am Bildschirm erscheinen. Die verschiedenen Stufen der Ausgabe werden dezidiert erläutert: Verwendet werden im Einzelnen die neuen Assembler-Befehle: XCHG, PUSH, POP, XOR, AND und SHR.

Neue Befehle und deren Bedeutung:

XCHG	<i>(to exchange → tauschen): Tauscht kompatible Registerinhalte oder Variablen mit Registerinhalten; der direkte Tausch zweier Variableninhalte ist nicht möglich (byte ptr oder word ptr können angewandt werden).</i>
PUSH	<i>(to push → drücken, schieben): Legt ein 16-Bit Registerinhalt oder 16-Bit-Variablen auf den Stack. Die Stack-Spitze wird um 2 erniedrigt. Ab Prozessor 80186 ist auch PUSHA möglich, wobei alle 16-Bit-Register in bestimmter Reihenfolge auf dem Stack abgelegt werden. Ab Prozessor 80386 kann der Befehl PUSHAD (→ push all doubleword registers) für 32-Bit-Systeme verwendet werden.</i>

POP	<i>(to pop → entnehmen, hochschnellen): Entnimmt den zuletzt per PUSH auf den Stack gepackten Wert und restauriert das entsprechende Register oder die Variable; korrespondiert mit dem PUSH-Befehl.</i>
XOR	<i>(→ exklusives Oder); gehört zu den logischen Operationen (vgl. Kapitel 9.3); wird hier nur zum Nullen eines Registerinhaltes benutzt.</i>
AND	<i>(→ logisches UND); nur wenn beim Vergleichen zweier Bitkombinationen an der gleichen Stelle jeweils eine „1“ steht, dann wird auch das Ergebnis 1. (vgl. Kapitel 9.3); eignet sich zum „Maskieren“ und „Demaskieren“ von Register- und Variableninhalten.</i>
SHR	<i>(→ SHift Right); innerhalb eines 8-,16- oder 32-Bit-Registers oder einer entsprechenden Variablen wird eine bestimmte Anzahl Bits, die beim 8086 in CL stehen muss, nach rechts verschoben. Wichtig: Der Inhalt von Bit 0 geht in das CF (Carry Flag) über und überschreibt dessen aktuellen Inhalt.</i>

Bei nachfolgender Betrachtung werden die einzelnen Schritte in sequenzieller Form gezeigt und können jeweils in konkrete Assembler-Befehle umgesetzt werden:

Schritt:	Aktion:	Assembler-Befehl	Inhalt AX
1.	AH- und AL-Register auf Stapel retten	PUSH AX	35 78
2.	AH- und AL-Register tauschen (weil zuerst die höherwertigen Bits ausgegeben werden)	XCHG AH,AL	78 35
3.	AH nullen	XOR AH,AH	00 35
4.	AX nochmals auf Stack	PUSH AX	00 35
5.	AL maskieren	AND AL,11110000B	00 30

6.	Shift 4 Bit nach rechts	MOV CL,4 SHR AL,CL	00 03
7.	30H addieren; AL nach DL und dort ausgeben	ADD AL,30H MOV DL,AL MOV AH,2 INT 21H	00 33
8.	Oberes Wort von Stapel nehmen	POP AX	00 35
9.	AL maskieren	AND AL,0F	00 05
10.	30H addieren; AL nach DL und dort ausgeben	ADD AL,30H MOV DL,AL MOV AH,2 INT 21H	00 35
11.	Ursprüngliches AX von Stack	POP AX	35 78
12.	AH löschen, AX auf Stack, AL maskieren	XOR AH,AH PUSH AX AND AL,11110000B	00 70
13.	Inhalt von AL nach rechts schieben	MOV CL,4 SHR AL,CL	00 07
14.	30H addieren; AL nach DL und dort ausgeben	ADD AL,30H MOV DL,AL MOV AH,2 INT 21H	00 37
15.	AX vom Stack holen	POP AX	00 78
16.	AL maskieren; 30H addieren; AL nach DL und dort ausgeben	AND AL,00001111B ADD AL,30H MOV DL,AL MOV AH,2 INT 21H	00 38

Nachstehend folgt der komplette Quellcode. Zu beachten ist, dass im einfachsten Fall keine data-Section definiert sein muss. Das Programm liefert lediglich die Ziffernkombination 3578 am Bildschirm. Es ist anzuraten das Programm wiederum im Einzelschrittmodus auszutesten und jeden Schritt nachzuvollziehen.

```

;*****
;
;* Programm Oss7.asm: Ausgabe AX-Register      *
;*
;*              nur code-Section              *
;*
;*****
;
ORG 100H

Bits 16

section .code      ; Beginn Code-Segment

MOV AX, 3578H      ; AX mit Anfangswert laden
PUSH AX           ;

;
;
;          ; vertausche AH und AL

XCHG AH,AL        ; nulle AH-Register
XOR AH,AH
PUSH AX           ; AX auf Stack legen
AND AL,11110000B  ; maskiere AL
MOV CL,4
SHR al,cl         ; in AL 4 Bits nach rechts (identisch mit Division /10hex)
;Zeichenausgabe über DL-Register

ADD AL,30H
MOV DL,AL
MOV AH,2
INT 21H

POP AX ; letzten gespeicherten Wert von AX von Stapel
AND AL,00001111B  ; maskiere AL
;Zeichenausgabe über DL-Register;
ADD AL,30H
MOV DL,AL
MOV AH,2
INT 21H

POP AX ; vorletzten gespeicherten Wert vom Stack holen
XOR AH,AH        ; nulle AH
PUSH AX          ; lege aktuellen AX-Inhalt auf Stack
AND AL,11110000B ; maskiere AL

```

```
MOV CL,4          ; lade Anzahl der zu schiebenden Zeichen          ;
SHR AL,CL          ; schiebe Anzahl von cl Zeichen in AL nach rechts
;Zeichenausgabe über DL-Register;
ADD AL,30H
MOV DL,AL
MOV AH,2
INT 21H

POP AX ; letzten gesicherten AX-Inhalt vom Stapel
AND AL,00001111B          ; maskiere AL

;Zeichenausgabe über DL-Register;
ADD AL,30H
MOV DL,AL
MOV AH,2
INT 21H

MOV AH,4CH
int 21H
```

- oss7.asm -

Hinweis: Natürlich erscheint die Ausgabe von Registerinhalten, gerade, wenn man bedenkt, dass man vielleicht sogar 32- oder 64-Bit breite Register verwendet, sehr aufwändig. In diesem Falle würde man auf die Vereinfachungsmöglichkeit durch Verwendung von Makros (siehe an anderer Stelle des Skriptums) zurückgreifen.

Ü9: Ausgabe von 8-Bit Hex-zahlen:

Im AL-Register stehe die Hexzahl A5. Diese beiden Hexzeichen sind zunächst nach DL zu kopieren und dort als Einzelzeichen am Bildschirm auszugeben.

- Ausgabe von Strings

Unter Strings versteht man alphanumerische Zeichenketten, die in der Informatik eine sehr große Rolle spielen. Diesem Themenkomplex ist an anderer Stelle viel Platz eingeräumt. An dieser Stelle soll es lediglich um die AUSGABE von Strings gehen.

Strings werden in der Data-Section deklariert. Wichtig ist, dass der String

⇒ eine Endekennung (\$) und

⇒ in der Regel den ASCII-Code für **CR** (Carrige Return → 13) und **LF** (Line Feed → 10) erhält.

In der Data-Section könnte die Deklaration einer String-Variablen z.B. wie folgt aussehen:

```
section .data
```

```
ueberschrift db "Assembler-Programmierung",10,13,"$"
```

Der String ist eine Kette von 8-Bit-Inhalten (jedes alphanumerische Zeichen wird durch ein Byte dargestellt). Nach der Zeichenkette, die in Anführungszeichen definiert wird, werden die ASCII-Codes für Line-Feed und CR (oder umgekehrt) und die Endekennung \$ angehängt. Damit ist der String zur Bildschirmausgabe vor-bereitet.

Zur Stringausgabe steht die Funktionsnummer 09 des INT 21H zur Verfügung. Die Referenz des Strings braucht nur noch dem DX-Register zugeordnet werden, um den String am Bildschirm zu zeigen.

Die Codierung oss8.asm gibt nacheinander 3 Stringinhalte in 3 aufeinanderfolgenden Zeilen aus:

```
*****
;
;* Programm oss8: Ausgabe Strings                                *
;*                                                                *
*****

ORG 100H

Bits 16

section .code

    ; Ausgabe Überschrift

    mov ah,9

    mov dx,ueberschrift

    int 21H

    ; Ausgabe Variable kapitel1

    mov ah,9                ; ggf. entbehrlich

    mov dx,kapitel1

    int 21H

    ; Ausgabe Variable kapitel2

    mov ah,9

    mov dx,kapitel2

    int 21H

    MOV AH,4CH

    int 21H

section .data

ueberschrift db "Assembler-Programmierung",10,13,"$"

kapitel1    db "Kapitel 1: Registeroperationen",10,13,"$"

kapitel2    db "Kapitel 2: Stack-Operationen",10,13,"$"
```

- oss8.asm -

Ü10: Ausgabe verschiedener Stringinhalte:

Definieren Sie insgesamt 4 String-Variablen in der Data-Section. Durch entsprechende Anordnung soll Ihr eigener Name und Adresse gespeichert und wiefolgt zur Anzeige gebracht werden:

Bert Osswald

Neutsch 21

64397 Modautal

Achten Sie auf die exakte Definition von LF und CR.

8. Kontroll- und Wiederholstrukturen

Letztendlich sind Kontroll- und Wiederholstrukturen für jede Programmiersprache konstitutiv, so auch für die Assemblersprachen. Der Unterschied zwischen Hoch- und Assemblersprache bringt jedoch einige signifikante Unterschiede.

8.1 Vergleichs- und Sprungbefehle

Man unterscheidet bedingte und unbedingte Sprünge. Unbedingte Sprünge entsprechen der GOTO-Anweisung in diversen Hochsprachen. Deren Anwendung widerspricht in Hochsprachen den Regeln der Strukturierten Programmierung. In Assembler sind Sprungbefehle unumgänglich. Aus diesem Grund lassen sich Assembler-Grobstrukturen auch vielfach gut mit Programmablaufplänen darstellen.

CMP- und JMP-Befehl:

- Der CMP-Befehl vergleicht die Inhalte von Registern, von Registern und Variablen oder von Registern mit Konstanten miteinander. Auch hier gelten wieder die Regeln, dass 2 Variableninhalte nicht direkt vergleichbar sind, dass man vielmehr Register dazwischenschalten sind. Auch sind hier ebenfalls die Regeln der Äquivalenz bei 8-Bit, 16-Bit oder 32-Bit-Systemen zu beachten:

.....

Variable1 db "x"

Variable2 dw 1324

Variable3 dd "HANS"

.....

cmp bx,word ptr Variable1

cmp ecx, Variable3

cmp dx,variable2

- der **jmp** (JUMP-Befehl) steuert eine Sprungmarke an. Man unterscheidet unbedingte und bedingte JMP-Befehle. Im Falle der unbedingten Sprunganweisung wird dem Befehl einfach eine Sprungmarke hinzugefügt:

JMP Marke

Oftmals wird der Sprung jedoch in Abhängigkeit eines Ereignisses getätigt. Die häufigsten Varianten des Sprungbefehls sind:

<u>Befehl</u>	<u>Bedeutung</u>	<u>Geprüfte Statusflags</u>
JA = JNBE	<i>Jmp if above (springe, wenn größer); identisch mit Jmp if not below or equal)</i>	CF=0 oder ZF=0
JAE = JNB	<i>Jmp if above or equal (springe, wenn größer oder gleich); identisch mit Jmp if not below)</i>	CF=0
JB = JNAE	<i>Jmp if below (springe, wenn niedriger); identisch mit Jmp if not above or equal)</i>	CF=1
JBE = JNA	<i>Jmp if below or equal (springe, wenn niedriger oder gleich); identisch mit Jmp if not above)</i>	CF=1 oder ZF=1
JC	<i>Jmp if carry (springe, wenn das Carry-Flag gesetzt)</i>	CF=1
JE = JZ	<i>Jmp if equal (springe, wenn gleich); identisch mit Jmp if zero)</i>	ZF=1
JNC	<i>Jmp if not carry (springe, wenn das Carry-Flag nicht gesetzt)</i>	CF=0
JNE=JNZ	<i>Jmp if not equal (springe, wenn nicht gleich); identisch mit Jmp if not zero)</i>	ZF=0
JCXZ	<i>Jmp if CX-Register zero (springe, wenn CX null);</i>	CX=0

Nachfolgendes Programm wendet verschiedene Sprungbefehle mit entsprechenden Sprungmarken an:

```
; Programmname: Oss9.ASM *****
;
;*****
;
;* Programm : Vergleichen verschiedener Operanden und Sprungmarken *
;
;* *
;* *
;*****

ORG 100H

Bits 16

section .CODE ; Beginn Codesegment

Start:
M1:

    xor ax,ax ; setze AX und BX auf 0
    xor bx,bx ; kürzer als MOV BX,0
    mov al,[op1] ; Inhalt Operand 1 nach AL
    mov bl,[op2] ; Inhalt Operand 2 nach BL
    cmp al,bl ; vergleiche AL und BL
    ja erstergroesser ; wenn erster größer dann Sprung
    jmp zweitergroesser

erstergroesser:

    mov dx,Meldung1 ; Ausgabefunktion Zeichenketten
    mov ah,9
    int 21H

weiter:    xor ax,ax ; setze AX und BX auf 0
           xor bx,bx
           mov al,[op3] ; Inhalt Operand 3 nach AL

           mov bl,[op4] ; Operand 4 nach BL
```

```
        cmp al,bl                ; vergleiche beide
        jl  zweitergroesser      ; wenn 2. kleiner
        jmp weiter1              ; unbedingter Sprung

zweitergroesser:                ; Sprungmarke
        mov dx, Meldung2
        mov ah,9
        int 21H

weiter1:
        xor ax,ax                ;Anweisungen für gleich grosse
        xor bx,bx                ; Inhalte
        mov ax,[op5]
        mov bx,[op6]
        cmp ax,bx
        je  gleichgross
        jmp ENDE

gleichgross:                    ; Sprungmarke
        mov dx, Meldung3
        mov ah,9
        int 21H

ENDE:                            ;Sprungmarke Programmende
        mov ah,4CH
        int 21H

section .Data
Meldung1 db 10,13,"erster Operand ist groesser$" ; Bildschirmausgaben
Meldung2 db 10,13,"zweiter Operand ist groesser$"
Meldung3 db 10,13,"beide sind gleich gross$"

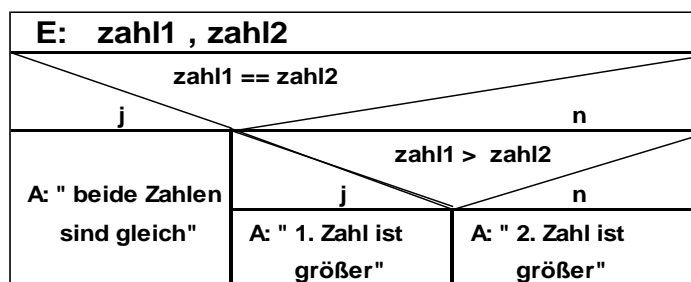
op1     db 5                    ; Operanden
```

```
op2  db 3                ;
op3  db "A"              ; hier mit unterschiedl. ASCII
op4  db "X"
op5  dw 1234H            ; WORD-Variablen
op6  dw 1234H
```

Obiges Program ist natürlich nicht besonders exakt, geschweige denn elegant. Es wird lediglich auf die vorgegebenen Operanden-Inhalte reagiert. Auch die Ausgaben der Zeichenketten auf dem Bildschirm werden später in ein Makro geschrieben, so dass sich die Länge des sichtbaren Quellcodes erheblich reduziert.

Im folgenden ist ein Programm zu entwickeln, das zwei einstellige Zahlen (dezimal) über Tastatur erwartet und über die obigen bedingten Sprung-Anweisungen miteinander vergleicht. Außerdem wird überprüft, ob die beiden eingegebenen Zeichen Ziffern zwischen 0 und 9 sind!

Struktogramm:



Wie vorstehendes Struktogramm zeigt, sollen die beiden eingegebenen einstelligen Zahlen auf Gleichheit überprüft werden, wenn diese verneint wird, soll mit „größer als“ abgefragt werden. Wenn diese 2. Bedingung ebenfalls verneint wird, dann muss automatisch die 2. Zahl größer sein.

```

;*****
;
;* Programm      : Eingabe zweier einstelliger Ziffern mit Gültigkeitsabfragen und Größenvergleich
;*
;
;Programmname: Oss10.ASM *****
;*****
;
;
;*****
;
%macro ausgabe 1                                ; Macro für Stringausgaben
mov dx,%1                                       ; Parameter: Anzahl Param
mov ah,9
int 21H
%endm

Org 100H
Bits 16
section .CODE

ORG 100H
Start:

Eingabeschleife1:                               ; bei Falscheingaben Sprungziel
    ausgabe Eingabe                             ; Eingabeaufforderung ausgeben
    mov ah,1
    int 21H
    cmp al,30H                                  ; wenn ASCII kleiner 30, dann ist es keine Ziffer
    jl Eingabeschleife1                        ; zurück zur Eingabe
    cmp al,39H                                  ; wenn ASCII größer 39, dann ist es keine Ziffer
    jg Fehler                                  ; zurück zur Eingabe

```

```
sub al,30H
mov [op1],al
jmp Eingabeschleife2
Fehler:
    ausgabe Fehlerm                ; Fehlermeldung mit Rücksprung zu E-schleife 1
    jmp Eingabeschleife1

Eingabeschleife2:                ; bei Falscheingaben Sprungziel
    ausgabe Eingabe                ; Eingabeaufforderung ausgeben
    mov ah,1                        ; Eingabe mit Bildschirmecho
    int 21H
    cmp al,30H                    ; wenn ASCII kleiner 30, dann ist es keine Ziffer
    jl Eingabeschleife2            ; zurück zur Eingabe
    cmp al,39H                    ; wenn ASCII größer 39, dann ist es keine Ziffer
    jg Fehler2
    sub al,30H                    ; Demaskierung
    mov [op2],al                  ; speichern
    jmp Vergleichen

Fehler2:                          ; Fehlermeldung mit Rücksprung zu E-schleife 2
    ausgabe Fehlerm
    jmp Eingabeschleife2

Vergleichen:                      ; vergleichen
    mov al,[op1]
    mov bl,[op2]                  ; Operanden stehen in 8-Bit-Registern
    cmp al,bl
    je gleich                    ; wenn 1. OP = 2. OP gehe nach GLEICH
    jg eingroesser                ; wenn der 1. > 2. dann Sprungziel EINGROESSER
    jmp zweigroesser              ; bleibt als Restmöglichkeit übrig

gleich:                          ; Sprungziele mit entsprechenden Stringausgaben
    ausgabe Meldung1
    jmp ende                      ; nach Ausgabe ans Programmende springen
```

```
einsgroesser:
    ausgabe Meldung2
    jmp ende                ; nach Ausgabe zum Programmende

zweigroesser:
    ausgabe Meldung3

ende:
    ;Programmende
    MOV AH,4CH
    int 21H

section .Data
Eingabe db 10,13,"Geben Sie eine Ziffer zwischen 0 und 9 ein!$" ; Meldungen f. Bildschirm
Fehlerm db 10,13,"Das war leider keine Ziffer! $"
Meldung1 db 10,13,"beide Zahlen sind gleich gross$"
Meldung2 db 10,13,"die erste Zahl ist groesser$"
Meldung3 db 10,13,"die zweite Zahl ist groesser$"

op1 db 0 ; 8-Bit-Speicher-Variablen
op2 db 0
```

Ü11: Testen Sie obiges Programm im Einzelschrittmodus aus und beachten Sie intensiv die Veränderung der Flag-Register!

Ü12: Nehmen Sie Bezug auf Programm Ü9 (S. 31) und erweitern Sie das Programm dergestalt, dass neben den

Ziffern 1 bis 9 auch die Hexzahlen A,B,C,D,E und F am Bildschirm gezeigt werden.

Ü13: Erweitern Sie obige Problemstellung auf die Eingabe von 2-stelligen Dezimalzahlen. Verwenden Sie dazu De-maskierungs- und ggf. Schiebebefehle. Auf Gültigkeitsprüfungen soll wegen zu hoher Komplexität des Programmes verzichtet werden. Wenn die beiden Dezimalzahlen erfasst wurden, sind sie wiederum in die beiden 8-Bit-Variablen zu speichern und größtmäßig zu vergleichen.

8.2 Schleifenkonstrukt

Ähnlich der Hochsprachen-Programmierung kann man auch in Assembler eine „fußgesteuerte“ oder „abweisende Schleife“ programmieren. Die Anzahl der Durchläufe muss zum Schleifenbeginn in CL (bzw. CX oder ECX) stehen. Während des Schleifen-Durchlaufs wird der Wert in CL automatisch um 1 vermindert. Durch die Anweisung **LOOP** und Angabe der Sprungmarke werden die Anweisungen im Schleifenkörper CL Mal ausgeführt.

Nachstehendes (Oss11.asm) Programm schreibt nacheinander die Großbuchstaben A,B,C,D und E auf den Bildschirm:

Anweisungsschritte:	Befehle:	Anmerkung:
Start:		
Festlegung des Anfangs-Hexwertes DX	MOV DX,40H	ASCII (dez) 41 = A
Anzahl der Schleifendurchläufe CX zuwei-	MOV CX,5	

sen		
Schleifendurchlauf mit automatischem CL-Dekrement		
DX-Wert in Schleife erhöhen	INC DX	1. Ausgabe ist somit Zeichen A
Ausgabe des relevanten ASCII-Zeichens	MOV AH,2 INT 21H	
Schleifenende	LOOP (Sprungmarke)	Läuft so lange CX>0

```

;*****
;
;* Programm OSS11.ASM: Schleifenkonstrukt mit LOOP
;*****

```

```

Org 100H

Bits 16

Section .CODE

Start:

    MOV cx,5

    MOV dx,40H

marke:

    inc dx

    MOV ah,2

    Int 21H

```

```

loop marke

ENDE:

    MOV AH,4CH

    int 21H

```

Ü14: Schreiben Sie ein Programm, das nacheinander alle ohne Rest durch 2 teilbaren Ziffern im Intervall zwischen 0 und 9 am Bildschirm ausgibt unter Verwendung des LOOP-Befehls.

9. Bit-Operationen

Gemäß der mathematischen Logik, werden auch in der Assembler-Programmierung logische Grundregeln verwendet. Man unterscheidet im wesentlichen AND, OR, XOR und NOT. Alle 4 Ausprägungen sollen kurz dargestellt werden:

Bei der **OR-Verknüpfung** (inklusive Oder) reicht das Vorhandensein einer 1 (bzw. eines Stromimpulses), damit das Ergebnis ebenfalls 1 wird. Folgende Tabelle stellt die möglichen logischen Kombinationen dar:

A	b	a OR b	
0	0	0	
1	0	1	
0	1	1	
1	1	1	

Die Oder-Verknüpfung (bzw. Oder-Schaltung) ist übrigens Grundlage der Addition im Dualsystem. Es muss lediglich dafür gesorgt werden, dass im Fall des Auftretens von Einsen in beiden Eingängen beim Ergebnis eine Null angehängt wird ($1 + 1 = 10$; was schlicht bedeutet, dass $1 + 1$ vom Ergebnis her 2 wird!). Laden wir z.B. das AL-Register mit dem Inhalt 05 und bringen wir den ODER-Befehl

OR AX,06

So erhalten wir als Ergebnis 7. Wie kommt dies zustande? Wir rechnen den Inhalt 05 dual um und kommen zum Ergebnis 0000 0101.

Das gleiche machen wir mit 06: 0000 0110.

Ergebnis: 0000 0111.

Immer dann, wenn Nullen untereinander stehen, wird auch das Ergebnis gleich 0, ansonsten, wenn wenigstens eine Eins vorkommt, dann wird auch das Ergebnis 1.

Im Fall der Addition ist natürlich eine Korrektur vorzunehmen, dass wenn 2 Einsen untereinander stehen, eine 0 als Ergebnis und eine 1 als Übertrag folgt, so dass das Additionsergebnis 0000 1011 → 11 Dezimal wird.

Bei der **XOR-Verknüpfung** (exklusives Oder, „entweder...oder“) wird das Ergebnis nur dann 1, wenn sich die beiden Operanden (Stimuli) unterscheiden:

A	b	a XOR b	
0	0	0	
1	0	1	
0	1	1	
1	1	0	

In der Assembler-Programmierung wird diese Verknüpfung sehr gerne verwendet, um einen Operanden oder Registerinhalt auf 0 zu setzen. Im Gegensatz zur Anweisung

MOV AL,0

benötigt **XOR AL,AL** weniger Ausführungszeit und wird, da Assemblerprogramme auch unter dem Primat der Code-Optimierung geschrieben werden, häufig angewandt.

Nachfolgende Codierung definiert einen Variableninhalt in der Variable **var**, transferiert diesen ins AL-Register, maskiert ihn im DL-Register und gibt ihn am Bildschirm aus.

Im 2. Schritt wird der gleiche Inhalt in AL auf 0 gesetzt (**per XOR AL,AL**), der Variable wiederum zugewiesen und ebenfalls am Bildschirm gezeigt:

```

;*****
;
;* Programm : Variableninhalt auf 0 setzen und rückspeichern      *
;
; Programmname: Oss12.ASM *****
;*****
;

ORG 100H

Bits 16

%macro ausgabe 1
mov dx,%1
mov ah,9
int 21H
%endm

section .CODE                ; Code-Segment

Start:

    ausgabe vorher

    mov al,[var]              ; Transfer Inhalt nach AL
    mov dl,al                 ; dito nach DL
    or  dl, 110000B           ; Maskierung im hex. ASCII

    mov ah,2                  ; Bildschirmausgabe
    int 21H

    ausgabe vorschub

    ausgabe nachher

    mov al,[var]              ; gleiche Zuweisung
    xor al,al                 ; Inhalt von AL wird "zeitkritisch" auf 0 gesetzt
    mov [var],al              ; neuer Inhalt in die Variable zurückspeichern
    mov dl,[var]

    or  dl, 110000B           ; Inhalt nach DL wegen Ausgabe

```

```

                                ; s.o.

mov ah,2
int 21H

                                ;Programmende

MOV AH,4CH
int 21H

section .Data                    ; Daten-Segment

var db 6
vorher db "Variableninhalt vorher $"
vorschub db 10,13,"$"
nachher db "Variableninhalt nachher $"

```

Bei der **NOT-Verknüpfung** werden die Bits innerhalb eines Ergebnisses einfach umgedreht.

Ergebnis	NOT Ergebnis
1	0
0	1

Bei der **AND-Verknüpfung** müssen beide Operanden 1 sein (elektrotechnisch fließt Strom), damit das Ergebnis a AND b ebenfalls 1 wird:

A	b	a AND b	
0	0	0	
1	0	0	
0	1	0	
1	1	1	

Programmiertechnisch kann man auf diese Art und Weis Teile eines 8-,16- oder 32-Bit Registers oder Operands löschen. Sollen z.B. nur die unteren 4 Bit des AX-Registers gelöscht werden, notiert man:

AND AX,FFF0H

Hier bedeutet F jeweils die duale Kombination 1111.

Ist in diesem Beispiel das AX Register mit dem Inhalt 4321 geladen, so bleibt nach Abarbeitung des Befehls **AND AX,FFF0H** 4320 im Register stehen.

Man kann hier noch differenzierter vorgehen: Wenn in einem 16-Bit-Register nur das 9. Bit interessiert (wobei zu beachten ist, dass das Zählen der Bits wiederum bei 0 beginnt!) und ausgelesen werden soll, dann kann folgende Anweisung dies bewerkstelligen:

AND AX, 0000 0010 0000 0000B

Diese Vorgehensweise eignet sich sehr gut zum Auslesen von Status-Bytes. Man kann sich z.B. vorstellen, dass die Einstellungen eines Druckers durch einzelne Schalter angezeigt werden. Dies könnte folgendermaßen aussehen:

Bit Nr.	7	6	5	4	3	2	1	0
Status	1	1	0	0	1	0	0	1
Bedeutung	Papier vorhanden	Farbkartuschenstatus	Local Disk-space > 10MB	Spool ein	Automatische Erkennung ein	beidseitig	Netzdruck	Eingeschaltet

Angenommen, dieser Druckerstatus würde geladen und ins BX-Register überführt, dann könnten die entsprechenden Einsen durch die Anweisung

AND BX,0FFH

sichtbar gemacht werden. Unser 16-Bit-Register „verweigert“ aber bekanntlich die Auskunft über Nullen und Einsen. Als Ergebnis wird lediglich C9 dargestellt. Es gibt jedoch Assemblerbefehle, die uns das gewünschte Ergebnis liefern.

Neuer Befehl:

RCL (Rotate Carry Left)

Bits werden links aus Register durch das Carry-Flag gedrückt, welches dadurch abgefragt werden kann!

```

;*****
;
;* Programm : Einsen innerhalb eines Registers Auslesen mit Hilfe des
;
;* RCL-Befehls
;
;Programmname: OSS13.ASM *****
;*****
;
ORG 100H

bits 16

section .CODE ; Code-Segment

; lege Code ab Speicherstelle 100 Hex ab

M1:

    mov al,7

    mov cx,8 ; 8 Bit sollen angezeigt werden

loop1: ; Schleifen-Sprungmarke

    rcl al,1 ; rotate left (Bits werden links aus AL-Register rotiert, unter Einbezie-
    call dual ; ung des Carry-Flags, das dann abgefragt werden kann ==> Aufruf der
    ; Subroutine DUAL

    loop loop1 ; Schleifenende: Wenn CX ungleich null, wiederhole die Schleife

```



```
ENDE:

    MOV AH,4CH

    int 21H

dual:                ; lokale Sprungmarken

push ax                ; Hauptregister auf den Stack
push bx
push cx
push dx

pushf                ; Statuswort der Flags auf den Stack retten
dec cl                ; cl um 1 reduzieren, da beim nullten Bit beginnt
mov dl,cl            ; Zeichen nach DL
add dl,30H           ; Maskierung: Verwandle in den HEX-ASCII-Code
mov ah,2             ; Funktion Ausgabe des Zeichens
int 21H

popf                ; Status-Wort (Flags) von Stack nehmen
jc eins              ; wenn Carry-Bit gesetzt, dann zur entsprechenden Anzeige springen
mov dx, nicht        ; Stringausgabe
mov ah,9
int 21H

jmp ende              ; wenn 0 ausgegeben wurde, dann ans Ende springen

eins:
mov dx, gesetzt      ; Stringausgabe bei Bit = 1
mov ah,9
int 21H

ende:                ; Ende-Sprungmarke

pop dx                ; allgemeine Register von Stack nehmen
pop cx
pop bx
pop ax
ret

section .data          ; Daten-Segment
gesetzt db " Bit ist 1 ",10,13,"$" ; Strings - für Ausgabe mit Zeilenvorschub
```

nicht db ". Bit ist 0 ",10,13,"\$"

Bezugnehmend auf die Ausführungen zum Flagregister (vgl. S. 12) kann das Statuswort des 8086 Bit für Bit ausgelesen werden. Man achte darauf, dass zu einem bestimmten Zeitpunkt das Flagregister per PUSH-Befehl auf den Stapel zu retten ist; danach kann es per POP-Befehl in ein allgemeines Register abgelegt werden. Dort kann in Anlehnung an obiges Programm (OSS9.ASM) das Status-Wort am Bildschirm ausgegeben werden. Die Belegung des Statuswortes sei hier nochmals gezeigt:

Bit Nr.	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Belegung	n	n	n	n	OF	DF	IF	TF	SF	ZF	n	AF	n	PF	n	CF

Beachten Sie in diesem Zusammenhang unbedingt die auf der Seite 12 gezeigten Zusammenhänge über die Wertigkeit der nicht belegten Bits!

Ü15: Schreiben Sie ein Programm, das das Statuswort des 8086 auf gesetzte bzw. ungesetzte Bits untersucht. Die Nummerierung der Bits soll am Bildschirm erscheinen.

Da es sich um 16 Bit dezimal handelt, soll die Nummerierung der Bits 10 bis 15 aus Einfachheitsgründen hexadezimal (A..F) erfolgen.

10. Stringoperationen

Alphanumerische Zeichenketten bezeichnet man in der Programmierung als Strings. In der Assembler-Programmierung werden sie auch als fest definierte Speicherbereiche gehandhabt. Man unterscheidet Byte-, Wort- und Doppelwort-Strings.

10.1 Deklaration von Strings

Strings werden im Daten- und Extrasegment gespeichert. Da im Small-Speichermode neben dem Codesegment nur das Datensegment existiert, ist die Unterscheidung eher theoretischer Natur, kann aber nicht einfach ignoriert werden.

Der Inhalt von Strings wird in Anführungszeichen gesetzt und das Ende der Zeichenkette wird durch ein Dollarzeichen (\$) symbolisiert. Folgende Deklarationen sind möglich:

.DATA

Beispiel	Bedeutung:
String1 db "Dies ist ein Inhalt\$"	Byte-String, der durch einen Inhalt vorbelegt ist
String2 db "",10,13,"\$"	Byte-String mit vorgegebenem Leer-Inhalt und LF und CR. Die Endekennung folgt.

Durch bestimmte String-Verarbeitungsbefehle für den 8086 und Folge-Prozessoren werden die auf Assembler-Ebene notwendige Anweisungsketten strukturiert und vereinfacht. Im Folgenden wird aber jeweils eine ausführliche indexorientierte Verarbeitung zum Vergleich gegenübergestellt. Im weiteren Zusammenhang spielen die Indexregister SI (Source-Index) und DI (Destinations-Index) eine große Rolle.

10.2 Stringausgabe

Wurde eine Zeichenkette durch Initialisierung im Speicher abgelegt, dann kann sie sehr einfach mit der DOS-Funktion AH=09 zur Ausgabe gebracht werden. Diese bereits weiter oben angewandte Ausgabefunktion erwartet lediglich den Offset der Stringvariable im DX-Register. Die dahinterstehende Mikro-Programmierung holt Zeichen für Zeichen nach DL und fragt, ob die Endekennung "\$" erreicht wurde oder nicht. Eine weitere Alternative stellt der Stringbefehl: LODS dar:

Folgendes Beispiel gibt einen initialisierten String auf drei verschiedene Arten am Bildschirm aus:

.DATA

String1 db "Ich bin ein String\$"

Mögliche Varianten:

<i>Per DOS-Funktions-nummer 09</i>	<i>Per „Mikrocode“</i>	<i>Per String-verarbeitung s-befehl LODS</i>	<i>Anmerkungen</i>
MOV DX, string1	MOV SI, string1	MOV SI, string1	Referenz des Strings in allen Fällen notwendig!
MOV AH,9H	AUSGABE: MOV DL, [SI]	AUSGABE: LODSB	LODSB lädt das nächste Zeichen automatisch nach AL
INT 21H	CMP DL, "\$"	CMP AL, "\$"	War es die Endekennung?
	JE ENDE	JE ENDE	Wenn ja, dann springe auf das Ende-Label
		MOV DL,AL	Bei LODS AL nach DL notwendig
	MOV AH,2 INT 21H INC SI JMP AUSGABE	MOV AH,2 INT 21H JMP AUSGABE	Funkt.-nummer 02 mit Interrupt INC bei LODS nicht notwendig

Wie oben zu sehen, gewinnt man durch die Anwendung des LODS-Befehls mindestens 2 Anweisungen. Er lädt automatisch das über den Index adressierte nächste Byte (oder Wort!) nach AL und inkrementiert SI automatisch. Es ist je nach Deklaration immer zwischen LODSB (Load String Byte) und LODSW (Load String Word) zu unterscheiden!

Hier nun die Codierungs-Varianten:

```

*****
;
;* Programm : Stringausgabe unter Verwendung des Index-Registers SI.      *
;*
;*
;Programmname: Oss15.ASM *****
*****
;

```

```

ORG 100H

Bits 16

Section .CODE                      ; Code-Segment

Start:

    mov si, string1                ; String-Offset nach SI

Ausgabe:

    mov dl,[si]                    ; Dereferenzierung von SI

    cmp dl,'$'                     ; War das Zeichen die Endekennung?

    je ENDE                        ; wenn ja, dann springe zum ENDE-Label

    mov ah,2                       ; Ausgabe des Zeichens

    int 21H

    inc si                         ; inkrementiere Index-Register

    jmp Ausgabe                    ; unbedingter Sprung zum Ausgabe-Label

ENDE:                               ;
Programmende

    MOV AH,4CH

    int 21H

section .Data                      ; Daten-Segment

string1 db "Ich bin ein String $"  ; Strings - für Ausgabe mit Zeilenvorschub

formfeed db 10,13,"$"

```

```

*****
;
;* Programm : Stringausgabe unter Verwendung des LODSB-Befehls *
;*
;*
;Programmname: Oss15a.ASM *****
*****
;

ORG 100H

Bits 16

section .CODE                      ; Code-Segment

ORG 100H                          ; lege Code ab Speicherstelle 100 Hex ab

Start:

```

mov si, string1	; String-Offset nach SI
Ausgabe:	
LODSB	; LODS-Befehl lädt Zeichen in AL-Register
Mov dl,al	; Kopieren nach DL zur Ausgabe
cmp dl,'\$'	; war das Zeichen die Endekennung?
Je ende	; wenn ja, dann springe ans Ende
mov ah,2	; Zeichenausgabe
int 21H	
jmp Ausgabe	
ENDE:	;
Programmende	
MOV AH,4CH	
int 21H	
section .data	; Daten-Segment
string1 db "Ich bin ein String '\$"	; Strings - für Ausgabe mit Zeilenvorschub
formfeed db 10,13,"\$"	

10.3 Eingabe von Strings

Auch hier bietet sich die indizierte Eingabe an. STOSB (Store String Byte) ist die Äquivalenz zu LODSB auf der Eingabeseite. Besonders zu beachten ist, dass STOS fest mit dem DI – Indexregister korrespondiert!

	<i>Per String-verarbeitung-befehl STOSB</i>	<i>Anmerkungen</i>
	MOV DI, string1	Offset des Strings nach DI, da ins <u>Extra-Segment</u> per STOSB gespeichert wird
	MOV AH,1 Int 21H STOSB	Zeicheneingabe per Bildschirmecho per DOS-INT Zeichen aus AL an die [DI]. Stelle im

			<i>String gespeichert</i>
		CMP AL,13	<i>Enter Taste?</i>
		JNE EINGABE	<i>Wenn ja, Eingabe nächstes Zeichen</i>
		MOV AL,"\$" STOSB	<i>Endekennung nach AL und wiederum an die [DI]. Stelle speichern.</i>

Verwendung des STOS – Befehls:

```

.*****
;

;* Programm : Stringeingabe unter Verwendung der Funktion 02H mit STOSB *
;
;* *
;

;Programmname: Oss16.ASM *****
.*****
;

%macro ausgabe 1 ; ausgabe macro einfache Stringausgabe
mov dx, %1
mov ah,9
int 21H
%endmacro

ORG 100H

Bits 16

Section .CODE ; Beginn Code-Segment

Start:

; initialisiere Daten- und Extra-Segment

```

```

; hier ist das Extra-Segment wirklich notwendig!

ausgabe meldung1

mov di, estring1          ; für STOSB muss das Zielindex-Register verwendet werden!

Eingabe:

mov ah,01H               ; Zeicheneingabe mit Bildschirmecho

int 21H

stosb                    ; speichert das in AL stehende Zeichen an die Stelle [DI]

cmp al,13                ; wenn Enter nicht gedrückt wurde

jne Eingabe              ; bedinter Sprung zum Eingabe-Label

mov al,"$"               ; letzte Eingabe (Enter) durch Ende-Kennung überschrieben

stosb                    ; wiederum durch STOSB-Befehl

fertig:

ausgabe formfeed         ; jetzt kann der String durch die Funktion 09H

ausgabe estring1         ; am Bildschirm ausgegeben werden!

ende:                    ; Programmende

    mov ah, 4cH

    int 21H

Section .DATA

meldung1 db "Erfassen eines Strings ueber Tastatur",10,13,"$"

formfeed db 10,13,"$"

estring1 db ".....$"

```

Ü16: Schreiben Sie ein Programm, welches einen Eingabestring mit Maximallänge 15 (dez) definiert. Tätigen Sie eine Eingabe per STOSB und bringen Sie die tatsächliche Länge des Strings zusammen mit dem String-Inhalt zur Ausgabe:

Bsp:

Eingabe einer Zeichenkette: **Assembler**

Ausgabe des Stringinhalts: **Assembler**

Länge der Zeichenkette: 9

10.4 Strings kopieren

Oft entsteht die Notwendigkeit einen Quell- (Ursprungs-)string in einen Zielstring zu überführen (kopieren). Dies geschieht wiederum über die beiden Index-Register SI und DI.

Über ein bedingtes Schleifenkonstrukt können komprimiert die Stringinhalte aller Längen kopiert werden. Zu beachten ist lediglich, dass die Endekennung (\$) dem letzten Zeichen im Zielstring angehängt wird:

```

;* Programm : Stringcopy unter Verwendung der beiden indexregister SI und DI
;*
;
; Programmname: Oss17.ASM *****

                                ; Konstante für CR-Tasten-ASCII-Code

%macro ausgabe 1                ; Macro für Textausgabe
mov dx, %1
mov ah,9
int 21H
%endmacro

ORG 100H

Section .code                   ; Beginn Codesegment
ORG 100H

ausgabe formfeed                ; Bildschirmausgaben
ausgabe meldung1
ausgabe quelle
ausgabe formfeed

mov si, quelle                  ; Referenz des Quellstrings nach SI
mov di, ziel                    ; Referenz des Zielstrings nach DI
copy:                          ; Label für Schleifenanfang
    cmp byte[si], "$"           ; war das adressierte Zeichen ein $ ?
    jz raus

```

```

mov dl, byte[si]                ; kopiertes Zeichen in 8-Bit-Register

mov byte[di],dl                ; von dort aus an Byte-Adresse im Zielstring
inc si                          ; erhöhe Source-Index
inc di                          ; erhöhe Destination-Index
jmp copy

raus:
mov byte[di],"$"                ; hänge Endekennung an
                                ; nach Ziel. Durch den rep-Befehl werden alle
                                ; Zeichen kopiert

ausgabe formfeed                ; Bildschirmausgaben
ausgabe meldung2
ausgabe ziel

ENDE:                           ; Programmende

    mov ah, 4cH
    Int 21H

Section .DATA
meldung1 db "Ausgabe des Quellstrings: ", "$"    ; Meldungen
meldung2 db "Ausgabe des kopierten Zielstrings: ", "$"
formfeed db 10,13,"$"                ; Deklaration formfeed/linefeed
quelle db "Das ist ein vordefinierter String$"    ; Quellstring (initialisiert)
ziel db "*****$"
; Zielstring: als Bereich mit 40 *

```

Ü17: Schreiben Sie ein MACRO für die Berechnung der Stringlänge des Quellstrings. Speichern Sie die Länge als 8-Bit-Variable ab und verwenden Sie diese als Schleifenobergrenze und programmieren Sie den Kopiervorgang per LOOP-Schleife.

10.5 Strings vergleichen

Oft benötigt man den Vergleich zweier Zeichenketten. Gerade bei der Eingabe eines Code- oder Passwortes kommt man nicht umhin, einen hinterlegten String mit einem einzugebenden zu vergleichen. Die Assembler-Sprache des 8086 stellt dazu den COMPS-Befehl zur Verfügung. Dieser prüft innerhalb eines Schleifenkonstruktes das Abweichen des wiederum über das SI-Register adressierten Quellstrings vom Zielstring. Beim Auftauchen der entsprechenden Ungleichheit wird unter anderem das Carry-Flag gesetzt. Durch Verwendung des JC (Jump if Carry)-Befehls kann man die Stelle der ersten Abweichung des Quell- vom Zielstring feststellen. Gleichsam ließen sich die Anzahlen der übereinstimmenden oder abweichenden Zeichen feststellen.

Im übrigen kann natürlich zunächst einmal die unterschiedliche Länge beider zu vergleichender Strings über die Längen-Variablen ermittelt werden. Erst wenn die Längen gleich sind, rentiert die zeichenweise Untersuchung beider Strings.

In nachfolgendem Programm OSS18.ASM wird von gleicher Stringlänge ausgegangen und die Untersuchung auf das erste sich unterscheidende Zeichen konzentriert:

```

;*****
;
;* Programm: Stringvergleich
;* Prog.Name: OSS18.ASM –
;*****

ORG 100H

Bits 16

%macro ausgabe 1 ; Ausgabemakro für Texte

mov dx, %1

mov ah,9

int 21H

%endmacro

%macro laengengecheck 1

mov si,%1

mov bx,si

weiter:

cmp byte[si], '$'

je rechne

inc si

jmp weiter

rechne:

```

```
mov ax,si
sub ax,bx
mov byte[q_len],al
%endmacro

section .code

laengengecheck quelle

mov si, quelle ; Offsets den Indexregistern zuweisen
mov di, ziel
mov cl, [q_len] ; Länge beider Strings nach CL
xor al,al ; AL nullen

loop1:
mov al,byte[si]
mov bl,byte[di] ; hier mit Schleifentest
cmp al,bl ; berechne die Stelle der 1. Ungleichheit
jne raus
inc si
inc di
loop loop1 ; Schleifenende

raus:
mov al,[q_len] ; berechne Stelle der 1. Ungleichheit
sub al,cl
mov byte[stelle],al ; sichere diese Stelle
inc byte[stelle] ; inkrementiere diese, da String beim
ausgabe formfeed ; nullten Zeichen anfängt
ausgabe meldung2 ; Ausgaben im Fall der Ungleichheit

jmp Ende

ausgabe meldung3

mov al,[stelle] ; Vorbereitung der Ausgabe für die
add al,30H ; berechnete Stelle
mov dl,al ; Ziffer am Bildschirm ausgeben
mov ah,2
int 21H
jmp Ende ; springe zum ENDE-Label

drueber:
```

```

ausgabe meldung1          ; wenn Strings identisch, dann Ausgabe
Ende:
    mov ah, 4cH
    Int 21H
Section .DATA
stelle db 0
meldung1 db "Beide Strings sind identisch: ", "$" ; Ausgabetexte
meldung2 db "Die beiden Strings sind unterschiedlich: ", "$"
meldung3 db "Unterschied ab Zeichen: $"

formfeed db 10,13,"$"
blank db " $"

          ; Variable zum Fixieren des Vor-
quelle db "ein vordefinierter String$" ; kommens des 1. Unterschiedes
ziel db  "ein fordefinierter String$" ; Quell- und Zielstrings deklarieren
q_len db 0

```

Ü18: Ein Passwort sei innerhalb eines Quellstrings fest vorzugeben (Höchstlänge: 10 Zeichen). Der Ausführende des Programms habe 3 mal die Möglichkeit der Eingabe des richtigen Passwortes. Wird das Passwort richtig erfasst, so ist die Meldung „Sie haben Zutritt zum System“, im anderen Fall: „Sie sind nicht autorisiert das System zu nutzen“ am Bildschirm auszugeben.

10.6 Makros

10.6.1 Einfache Makros

10.6.2 Makros mit Parameterübergabe

10.7 Definition und Implikation von Makrobibliotheken

10.8 **Prozeduren**10.8.1 **Near- und Far-Prozeduren**

```

;*****
;
;* Programm: Aufruf eines externen Unterprogrammes per CALL - Befehl
;
;* Prog.Name: BAOSS10a.ASM -
;
;*****
;
;*****
;
;* : Aufrufprogramm bei 2 Quellcodes
;
; Bildschirmausgabe, dann
;
; Aufruf U1 (Quellcode in BAoss10u.asm) *
;
;* *
;
;*
;*****
;
ausgabe macro puffer
mov dx,offset puffer
mov ah,9H
int 21H
endm

Public Meldung2 ; Meldungen im Unterprogramm als
; public ausweisen

Public formfeed

DATA_seg segment public ; Data-Segment Beginn
; hier vollständige Segment-Definition nötig
; deren Name ist aber flexibel

; Anweisung public notwendig, weil mehrere
; mehrere Module ein einziges Datensegment
; nutzen

Meldung db "Hallo ich bin im Aufrufprogramm",10,13,"$"
Meldung2 db "und sehr motiviert$"
formfeed db 10,13,"$"
taste db "Taste druecken!$"

```

```

beenden db "Aufrufprogramm beendet!$"

data_seg ends                ; Ende Daten-Segment

CODE_seg segment public      ; Beginn Code-Segment

assume cs:code_seg, ds:data_seg ; Zuweisung der Segmente zu den
                                ; Segmentregistern

ORG 100H

    EXTRN u1:near             ; externes Unterprogramm, aber in der
                                ; gleichen EXE-Datei

Start:

    mov ax, DATA_seg         ; Datensegment initialisieren

    mov ds,ax

    ausgabe meldung

    ausgabe meldung2

    ausgabe formfeed

    call u1                   ; an dieser Stelle Aufruf des Unterprogrammes
                                ; in BAOSS10u

    ausgabe meldung

    ausgabe taste

    ausgabe formfeed

    ausgabe beenden

    mov ah,8                  ; Tastendruck zum Beenden des Aufruf-
                                ; programmes

    int 21H

Ende:

    mov ah,4CH

    int 21H

code_seg ends

    END Start

```

```

;*****
;
;* Programm: Unterprogramm per CALL aufgerufen :
;
;* Prog.Name: BAOSS10u.ASM -
;*****

```

```

;*****
;
;
;
;*      aufgerufen von BAOSS10a      *
;
;*****
;
;
data_seg segment public                ; Datensegment mit gleichem Namen
;                                     ; wie in Aufrufprogramm
;
;                                     ; public hier notwendig
Meldung db "Ich bin im Unterprogramm",10,13,"$" ; Zeichenkette, nur im Unterprogramm
;
extrn Meldung2:Byte                   ; externe Meldung (Byte) verwendet
;
extrn formfeed:byte                   ; auch formfeed in Aufrufprogramm
;
data_seg ends                        ; Ende des Datensegments
;
CODE_seg  segment public
;
assume cs:code_seg, ds:data_seg       ; Segment-Register-Zuweisungen
;
;
u1 proc near                          ; Prozedur:
;
    ausgabe meldung
;
    ausgabe meldung2
;
    ausgabe formfeed
;
    ret                               ; ret – Befehl sorgt für „Aufräumen“
;                                     ; des Stack
;
u1 endp
;
code_seg ends
;
;                                     ; hier nur END-Anweisung ohne weiteres
;                                     ; Symbol

```

10.8.2 Parameterübergabe bei Prozeduren

10.9 Makros oder Prozeduren?

11 Adressierungsarten in der Assembler-Programmierung

Die 80x86-Prozessoren stellen eine Reihe von Adressierungsarten zur Verfügung. Der Großteil der Adressierungsarten bezieht sich auf Speicheroperanden. Davon weichen lediglich zwei Arten ab, die zuerst aufgeführt werden. Bestimmte Adressierungsarten bzw. Varianten dieser Arten stehen erst bei neueren Prozessoren als dem 8086 bereit. Diese sind im einzelnen

12.1 Registeradressierung

Hier sind die Operanden in Registern enthalten. Z.B.:

```
MOV bx, ds
PUSH es
XCHG ah, al
```

12.2 Unmittelbare Adressierung

Bei dieser Adressierungsart wird ein numerischer Wert einem Register zugewiesen bzw. mit dem Inhalt eines Registers abgeglichen. Bei Befehlen mit mehr als einem Operanden ist es in aller Regel nicht möglich, mehr als einen Operanden unmittelbar anzugeben. Z.B.:

```
mov ax, 0BD01H
or cl, A0h
```

12.3 Speicheradressierung

Speicheroperanden werden durch eine Kombination der Elemente Basis, Index, Skalierung und Displacement gebildet, wobei bestimmte Elemente optional sind. Basisregister kann eines der Register AX, BX, CX, DX, BP, SP, DI oder SI bzw. die entsprechende 32-Bit-Variante sein. Als Index kann eines der Register AX, BX, CX, DX, BP, DI oder SI (bzw 32-Bit-Varianten) sein, nicht (E)SP. Die Skalierung kann nur in Zusammenhang mit einem Index benutzt werden. Dabei wird der Index mit einem Skalierfaktor von 1, 2, 4 oder 8 multipliziert. Beim Displacement handelt es sich um einen 8-Bit oder 32-Bit-Wert, der dem Befehl unmittelbar folgt. Das Displacement spielt hier die Rolle eines Offsets. Es ergeben sich die folgenden Adressierungsformen.

Direkt:

```
MOV ax, word ptr [500H] ; ab Adresse 500 ein Wort nach AX lesen
```

Register-Indirekt:

```
MOV ax, [si] ; Adresse steht in einem allg. oder Indexregister
```

Based:

MOV ax,[di + 12] ; Adresse = Inhalt allg. Registers + Displacement

Indexed:

MOV ax,Bereich[si] ; Adresse = Inhalt Indexregisters + Displacement

Based Indexed:

MOV bx,[di][bx] ; Adresse = InhaltIndexregisters +Inhalt allg.
;Register: Wirkt sich aus wie Addition beider Register

Bsp. Adressierungsarten

```

,*****
,* Programm Adressierungsarten          *
,* Prog.Name:                          *
,*****
,

.MODEL SMALL

.Data

Bereich db 4,3,7,9,6,9,7,8

Buchstaben db "A","X","C","$"

Myword dw ?

.CODE

ORG 100H

Start:

M1:   MOV AX,@Data           ; Registeradressierung

      MOV DS,AX              ; Registeradressierung

      PUSH DS                ; Registeradressierung

      POP ES                 ; Registeradressierung


      MOV ax, word ptr [ds:01H] ; direkte Speicheradressierung

      AND ax,OFH              ; unmittelbare Adressierung

      MOV si,5

      MOV bx,[si]              ; Register indirekte Adressierung

      MOV al,Buchstaben[si]    ; Indexed Adressierung

      MOV ax, word ptr Buchstaben[si] ; gleiches Ergebnis

```

```
MOV si,1
MOV bx,3
MOV al,Bereich[si][bx]      ;Based Indexed ==> beide Indices werden addiert

ENDE:
MOV AH,4CH
int 21H
END Start
```

12 Dateiverarbeitung in Assembler

13 Systemprogrammierung

13.1 Systemparameter feststellen und einstellen

13.2 TSR-Programmierung

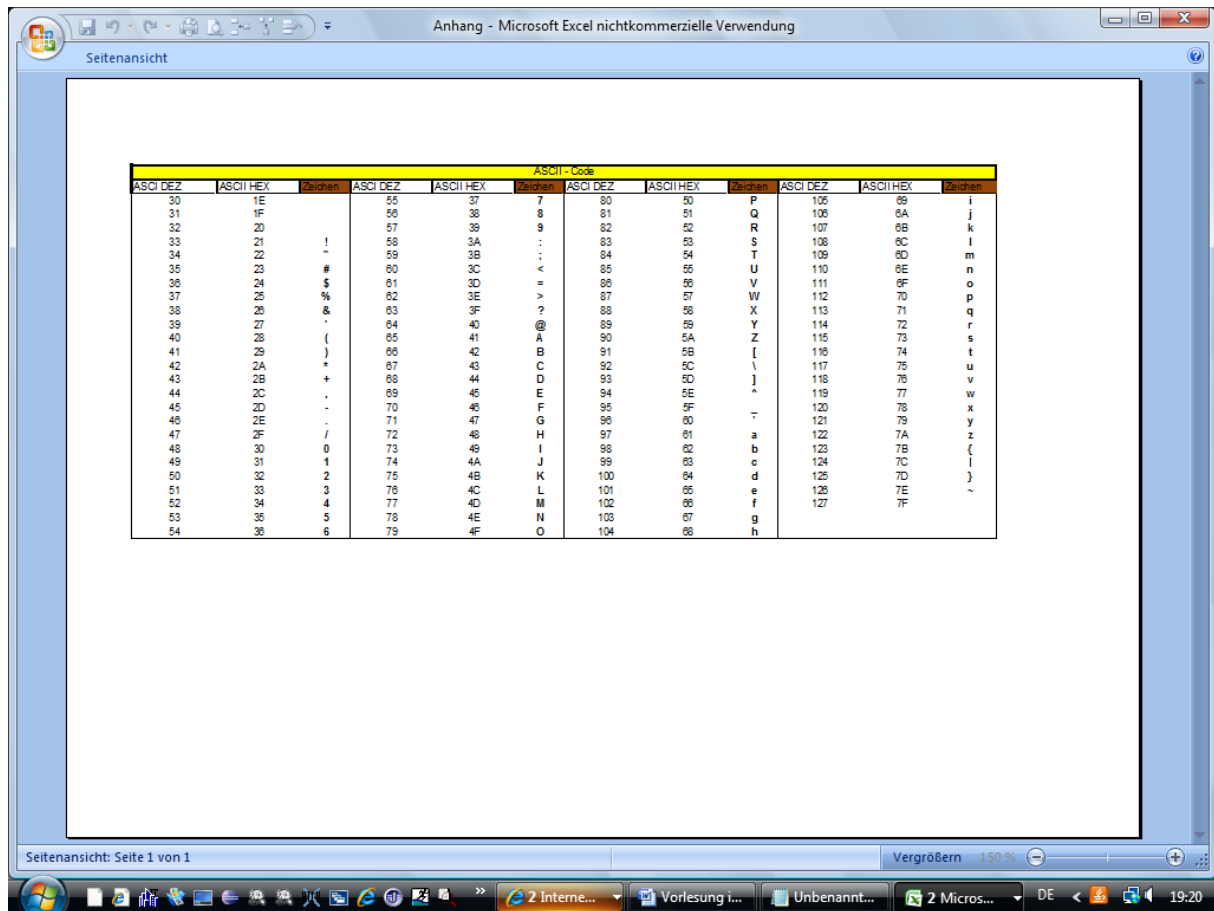
III. Hochsprachen und Assembler

I. 32-Bit-Programmierung

14 80386-, 80486-, MMX- und Pentium-Befehle

15 WINDOWS-Assembler-Programmierung

II. Anhang:



ASCII DEZ	ASCII HEX	Zeichen	ASCII DEZ	ASCII HEX	Zeichen	ASCII DEZ	ASCII HEX	Zeichen	ASCII DEZ	ASCII HEX	Zeichen
30	1E		55	37	7	80	50	P	105	69	i
31	1F		56	38	8	81	51	Q	106	6A	j
32	20		57	39	9	82	52	R	107	6B	k
33	21	!	58	3A	:	83	53	S	108	6C	l
34	22	"	59	3B	;	84	54	T	109	6D	m
35	23	#	60	3C	<	85	55	U	110	6E	n
36	24	\$	61	3D	=	86	56	V	111	6F	o
37	25	%	62	3E	>	87	57	W	112	70	p
38	26	&	63	3F	?	88	58	X	113	71	q
39	27	'	64	40	@	89	59	Y	114	72	r
40	28	(65	41	A	90	5A	Z	115	73	s
41	29)	66	42	B	91	5B	[116	74	t
42	2A	*	67	43	C	92	5C	\	117	75	u
43	2B	+	68	44	D	93	5D]	118	76	v
44	2C	,	69	45	E	94	5E	^	119	77	w
45	2D	-	70	46	F	95	5F	_	120	78	x
46	2E	.	71	47	G	96	60	`	121	79	y
47	2F	/	72	48	H	97	61	a	122	7A	z
48	30	0	73	49	I	98	62	b	123	7B	{
49	31	1	74	4A	J	99	63	c	124	7C	
50	32	2	75	4B	K	100	64	d	125	7D	}
51	33	3	76	4C	L	101	65	e	126	7E	~
52	34	4	77	4D	M	102	66	f	127	7F	
53	35	5	78	4E	N	103	67	g			
54	36	6	79	4F	O	104	68	h			

