Werner Rockenbach
odswer@uni-koblenz.de
Herzog-Johann-Gymnasium
55469 Simmern/Hunsrück

Maschinennahe Programmierung
(Integrierter Assembler von Turbo Pascal)

Der integrierte Assembler von Turbo Pascal erlaubt es, 8086-, 80286-Anweisungen direkt in den Pascal-Quelltext einzufügen. Die Syntax des integrierten Assemblers entspricht weitgehend der Syntax von Turbo Assembler bzw. Microsofts Macro-Assembler. Sein Befehlssatz umfaßt alle Befehle der 8086- und 80286-Prozessoren sowie die meisten Turbo-Assembler-Operatoren. Bei der maschinennahen Programmierung orientiert man sich direkt am jeweiligen Prozessor. Da die Intel-Prozessoren 8086, 8088, 80186, 80286, 80386 und 80486 alle den Befehlssatz des Prozessors 8086 verstehen, gelten alle Befehle für die 80x86-Prozessoren.

1. Modell Personalcomputer PC

In der CPU (Central Processing Unit) werden Register als spezielle Bausteine bereitgestellt, um Daten und Befehle zu verarbeiten.

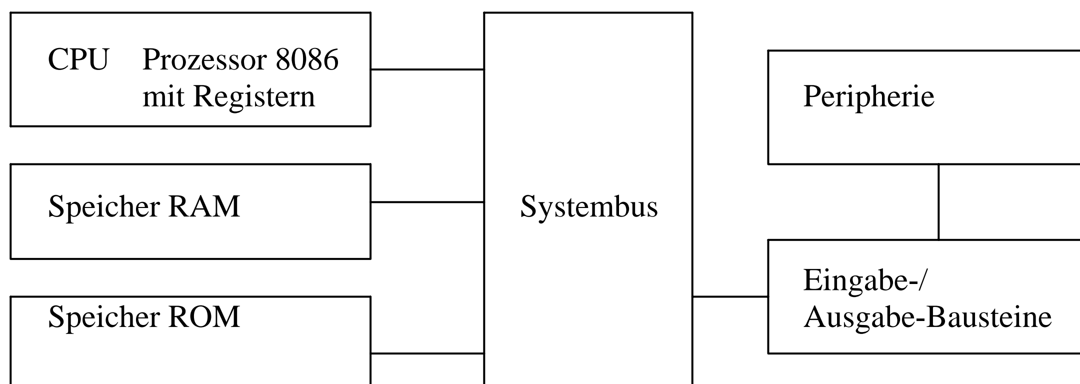
Im RAM (Random Access Memory) werden Daten und Befehle abgelegt, auf die im Zuge der Verarbeitung wiederholt direkt zugegriffen werden muss. Als flüchtiger Speicher geht der Inhalt des RAM beim Abschalten des Computers verloren.

Im Speicher ROM (Read Only Memory) wird die Information nur zum lesenden Zugriff bereitgestellt. Als Permanentspeicher bleibt die Speicherung dauernd erhalten.

Die Peripherie umfaßt die Eingabegeräte (z.B. Tastatur, Maus, Lichtgriffel), Ausgabegeräte (z.B. Bildschirm, Drucker) und Externspeicher (z.B. Disketteneinheit, Festplatteneinheit).

Über die entsprechenden Schnittstellen bzw. Gerätetreiber wird die Verbindung zwischen externen Einheiten (Peripherie) und internen Einheiten kontrolliert.

Der Systembus umfaßt alle Leitungen zur Übertragung von Daten, Befehlen und Adressen.



2. Register des Prozessors 8086

Die Register sind zwei Byte (16 Bit, ein Wort) breit, auf dem Prozessorchip untergebracht und zum schnellen, spezialisierten Zugriff vorgesehen.

2.1. Allzweckregister AX, BX, CX, DX

Akkumulator (AX), Basisregister (BX), Zählregister (CX) und Datenregister (DX) werden als Allzweckregister bezeichnet. Bei allen Registern des 8086-Prozessors handelt es sich um 16-Bit-Register. Die vier Allzweckregister lassen sich zusätzlich in zwei 8-Bit-Register unterteilen und ansprechen.

AX (Accumulator)	AL (Accumulator low)	AH (Accumulator high)
BX (Base-Register)	BL (Base-Register low)	BH (Base-Register high)
CX (Count-Register)	CL (Count-Register low)	CH (Count-Register high)
DX (Data-Register)	DL (Data-Register low)	DH (Data-Register high)

AH	AL	AX	Akkumulator (Addition, Subtraktion, Eingabe, Ausgabe)
BH	BL	BX	Basisregister (Zeiger auf Basis (Data-Segment))
CH	CL	CX	Zählregister (Schleife, Shift, Rotation)
DH	DL	DX	als Erweiterung für die Daten von AX

Der Akkumulator (AX-Register) wird hauptsächlich bei allen arithmetischen Operationen eingesetzt. Standardmäßig muss es bei der Multiplikation und Division eingesetzt werden. Das Register sollte aber auch bei der Addition, Subtraktion sowie bei logischen Vergleichen eingesetzt werden, da arithmetische Aufgaben mit diesem Register effizienter bearbeitet werden können.
Das Basis-Register (BX-Register) erlaubt es, direkt auf den Arbeitsspeicher zuzugreifen. Es wird als Zeiger auf den Inhalt des Speichers verwendet,
Das Count-Register (CX-Register) ist ein 16-Bit-Register zur Kontrolle von Programm-schleifen. Schleifen können effizient mit dem LOOP-Befehl formuliert werden. Dieser Befehl verwendet standardmäßig das CX-Register und subtrahiert von dessen Inhalt den Wert eins. Nach dieser Operation wird das CX-Register mit Null verglichen. Ist CX ungleich Null, wird zu dem im LOOP-Befehl angegebenen Namen gesprungen. Ist das CX-Register gleich Null, geht es mit dem folgenden Befehl nach LOOP weiter.
Das Daten-Register (DX-Register) wird hauptsächlich in Verbindung mit 32-Bit-Multiplikations- und Divisionsoperationen genutzt. Das Register wird dann zusammen mit dem AX-Register verwendet.

2.2. Adressregister SP, BP, SI, DI

Der Stackpointer (SP) zeigt stets auf die aktuelle Position im Stacksegment. Die physikalische Adresse zum Zugriff auf das Stacksegment wird durch das Registerpaar SS:SP dargestellt. Wenn Unterprogramme aufgerufen werden, dann speichert das System den Inhalt des Befehlszeigers (IP) auf dem Stack ab, um nach der Abarbeitung des Unterprogramms wieder an der richtigen Speicherstelle im Programm weiterarbeiten zu können. Diese Speicherstelle ist der nächste Befehl nach dem Unterprogrammaufruf.

Der Stack oder Stapel ist ein Zwischenspeicher, mit dem der Prozessor und der Programmierer kurzzeitig Daten zwischenspeichern kann. Das Stacksegment kann nur wortweise (2 Byte) beschrieben (PUSH) oder gelesen (POP) werden. Der Stack arbeitet nach dem LIFO-Prinzip (last in - first out). Mit dem Befehl PUSH wird ein Registerinhalt auf den Stack gelegt. Dabei wird der Stackpointer um den Wert zwei verringert. Mit dem Befehl POP wird ein Wert vom Stack geholt. Dabei wird der Stackpointer um den Wert zwei erhöht. Der Stack wächst also von den hohen zu den niedrigen Adressen.
Will man im Stacksegment auf zwischen gespeicherte Werte oder Parameter zugreifen, so wird der Base-Pointer (BP) benötigt. Wie die anderen Zeige- und Indexregister BX, SI und DI kann auch BP als Index verwendet werden.
Die beiden Indexregister SI (Source-Index) und DI (Destination-Index) unterstützen als Offsetregister die Adressierung im Datensegment und Extrasegment.

Register		
SP	SP	Register für Stapel-Zeiger (Stack-Pointer)
BP	BP	Basisregister (Stacksegment, Base-Pointer)
SI	SI	Indexregister (Quellzeiger, Source-Index)
DI	DI	Indexregister (Zielzeiger, Destination-Index)

2.3. Befehlsregister (IP), Statusregister

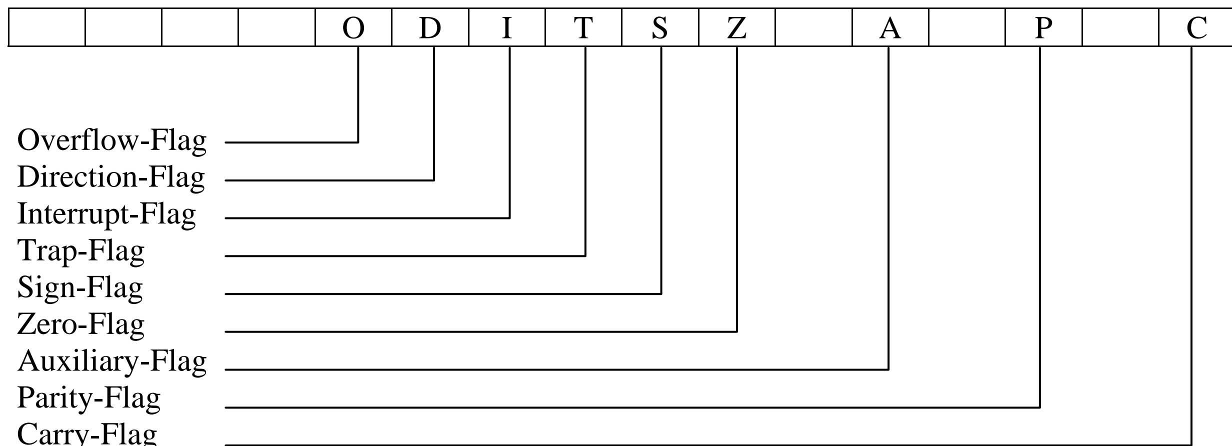
Der Befehlszeiger (IP), als Instruction-Pointer oder auch als Programm-Counter bezeichnet, adressiert in Verbindung mit dem Codesegment den nächsten auszuführenden Maschinenbefehl. Wird das Programm der Reihe nach Befehl für Befehl abgearbeitet, so enthält der Befehlszeiger den Offset der Speicheradresse, die den nächsten auszuführenden Maschinenbefehl beinhaltet. Mit dem Codesegment zusammen ergibt das Registerpaar CS:IP eine 32-Bit-Adresse, deren Inhalt beim nächsten HOLEN-Zyklus in das Befehlsregister geladen wird. Der Befehlszeiger zeigt damit immer wieder auf den nächsten im Codesegment befindlichen Maschinenbefehl. Bei einem Unterprogrammaufruf wird eine andere Speicheradresse in den Befehlszeiger geladen. Der Befehlszeiger kann im Gegensatz zu den anderen Registern nicht vom Programmierer angesprochen werden. Nur durch Sprungbefehle und Unterprogrammaufrufe kann der Befehlszeiger verändert werden.

Register		
IP	IP	Befehlsregister (Programmzähler, Instruction-Pointer)
Flags		Statusregister mit 8 Flags (O, D, I, T, S, Z, A, P, C)

Das Statusregister (Flagregister) zeigt Informationen über das Ergebnis eines Maschinenbefehls an. Das Statusregister ist ein 16-Bit-Register, von dem 9 Bits (Flags) benötigt werden. Steht ein Flag auf 1, so trifft der Zustand zu, den das Flag vertritt. Steht es auf 0, so trifft der Zustand nicht zu.
Diese 9 Flags unterteilen sich in 6 Statusflags und 3 Kontrollflags. Die restlichen Flags werden von den Nachfolgeprozessoren des 8086 benutzt.

Statusflags: Carry-, Parity-, Auxiliary-, Zero-, Sign-, Overflowflag
Kontrollflags: Trap-, Interrupt-, Directionflag

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



2.4. Flags

2.4.1 Carry-Flag (CF)

CF = 1 Wertebereich überschritten
Das Carry-Flag (BitNr. 0) wird gesetzt, wenn nach einer Addition oder Subtraktion der Wertebereich in einem Register überschritten wird. Dieses Flag zeigt somit den Übertrag aus dem höchstwertigen Bit an. Findet ein Übertrag statt, so steht das Carry-Flag auf 1. Der durch das Carry-Flag bedingte Sprung heißt jump on carry = jc (springe, wenn Carry-Flag gesetzt, d.h. CF = 1), bzw. jump not carry = jnc (springe, wenn Carry-Flag nicht gesetzt, d.h. CF = 0). Das Carry-Flag kann durch den Befehl CLC gelöscht (CF = 0) und durch STC gesetzt (CF = 1) werden.

2.4.2. Parity-Flag (PF)

PF = 1 Ergebnis einer Operation enthält in den niederwertigen 8 Bits eine gerade Anzahl an gesetzten Bits
Das Parity-Flag (BitNr. 2) dient zur Fehlerprüfung bei der Datenübertragung über die serielle Schnittstelle. Das Flag ist auf 1, wenn das Ergebnis einer Operation in den niederwertigen 8 Bits eine gerade Anzahl an gesetzten Bits enthält. Bei einer ungeraden Anzahl von gesetzten Bits wird das Parity-Flag auf 0 gesetzt. Der durch das Parity-Flag bedingte Sprung heißt jump on parity = jp (springe, wenn Parität gerade, d.h. PF = 1), bzw. jump not parity = jnp (springe, wenn Parität ungerade, d.h. PF = 0).

2.4.3. Auxiliary-Flag (AF)

AF = 1 Übertrag aus Bit 3 nach Bit 4 bei 8-Bit-Operationen
Dieses Flag ist dem Carry-Flag ähnlich. Es wird mit dem Hilfsübertrag ein Übertrag von Bit 3 nach Bit 4 bei einer 8-Bit-Operation angezeigt. Bei einem Übertrag steht AF auf 1. Der Hilfsübertrag wird hauptsächlich in der BCD-Arithmetik verwendet.

2.4.4. Zero-Flag (ZF)

ZF = 1 Ergebnis einer Operation ist null
Das Zero-Flag zeigt nach einer Operation an, ob das Ergebnis null ist oder nicht. Beim Ergebnis 0 steht das Zero-Flag auf 1.

Das Flag wird mit dem Schleifenbefehl LOOP eingesetzt, bei Vergleichsbefehlen und allen arithmetischen Operationen. Bei einem Vergleich von Zahlen werden diese intern subtrahiert, ohne die jeweiligen Register zu verändern. Ist das Ergebnis der Subtraktion null, so sind beide Zahlen gleich groß, das Zero-Flag wird gesetzt.

2.4.5. Sign-Flag (SF)

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

2.4.6. Overflow-Flag (OF)

OF = 1 Vorzeichenbit zerstört
Wenn bei einer arithmetischen Operation ein Übertrag auf das höchstwertige Bit erfolgt, wird das Overflow-Flag auf 1 gesetzt. Dies ist nur bei Werten mit Vorzeichen von Bedeutung. Das Overflow-Flag wird also gesetzt, wenn das Vorzeichenbit durch einen Übertrag verändert wird.

2.4.7 Trap-Flag (TF)

TF = 1 Nach jedem Befehl rufe den Interrupt 1 auf, und verzweige in die dazugehörige Unterbrechungsroutine.
Das Trap-Flag versetzt den Prozessor in den Einheitsschrittmodus. Durch das Trap-Flag wird der Prozessor angewiesen, nach jedem Befehl im Programm den Interrupt 1 aufzurufen.

2.4.8 Interrupt-Enable-Flag (IF)

IF = 1 Alle Unterbrechungen zugelassen
IF = 0 Keine maskierbaren Unterbrechungen zugelassen
Der Prozessor kann durch äußere Einflüsse (Interrupts) bei der Abarbeitung eines Programms unterbrochen werden. Gesetzt wird das Interrupt-Enable-Flag mit STI, gelöscht wird es mit CLI.

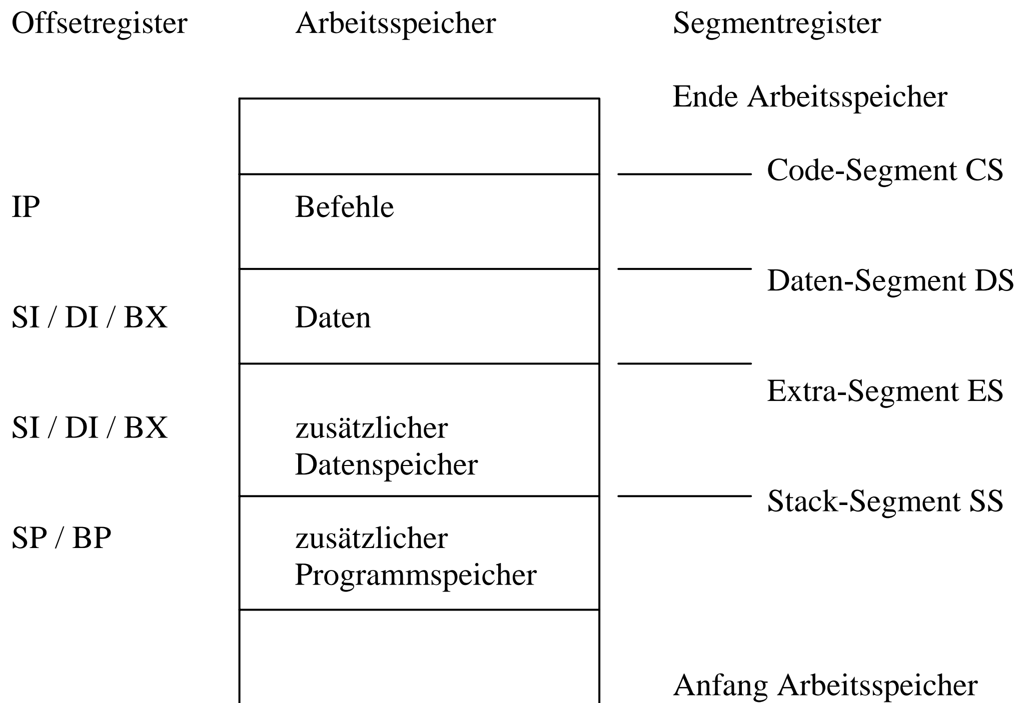
2.4.9. Direction-Flag (DF)

DF = 1 Stringverarbeitung nach aufsteigenden Adressen (SI und DI werden erhöht)
DF = 0 Stringverarbeitung nach absteigenden Adressen (SI und DI werden verringert)

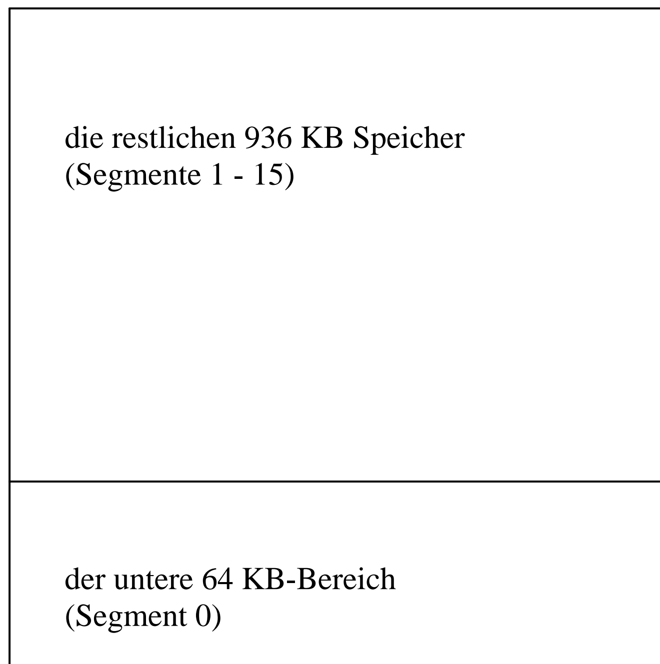
3. Speicheraufbau

Ein Hauptspeicher bzw. RAM von 1 MB (Megabyte) hat 1048576 Speicherplätze, die von 0 - 1048575 bzw. 0h - FFFFFh durchnumeriert sind. Der RAM ist in 16 Speicherbereiche unterteilt, die jeweils 64 KB umfassen und als Segmente bezeichnet werden.
Der Prozessor holt sich Daten und Anweisungen aus dem Arbeitsspeicher. Damit der Prozessor zwischen dem Anweisungsteil mit den Maschinenbefehlen und dem Datenteil mit den Variablen unterscheiden kann, werden die Daten in verschiedene Speicherbereiche getrennt. Diese Speicherbereiche werden als Segmente bezeichnet. Maschinenbefehle werden im Code-Segment, die Variablen und Konstanten im Daten-Segment gespeichert. Wenn der

Prozessor nun bestimmte Daten aus einem Segment lesen möchte, greift er dazu auf das dazugehörige Register zu. Jedem Segment wird standardmäßig ein Register zugeordnet.
Die physikalische Adresse wird aus dem Segment und dem Offset gebildet. Das Segment gibt die Basis an und das Offset den Adressabstand von der Basis zur jeweiligen Speicherstelle.



hexadezimal	dezimal
FFFFF	1048575
FFFFE	1048574
FFFFD	1048573
FFFFC	1048572
...	
...	
...	
10002	65538
10001	65537
10000	65536
0FFFF	65535
0FFFE	65534
0FFFD	65533
...	
...	
...	
00001	1
00	0



4. Beispiele

4.1. Wertzuweisung (Addition/Subtraktion)

```
procedure wertzuweisung;
var a, b, c : word;
begin
        a := 20;
        b := 12;
        c := a + b;
        a := a - b;
        Ausgabe (a, b, c)
end.
```

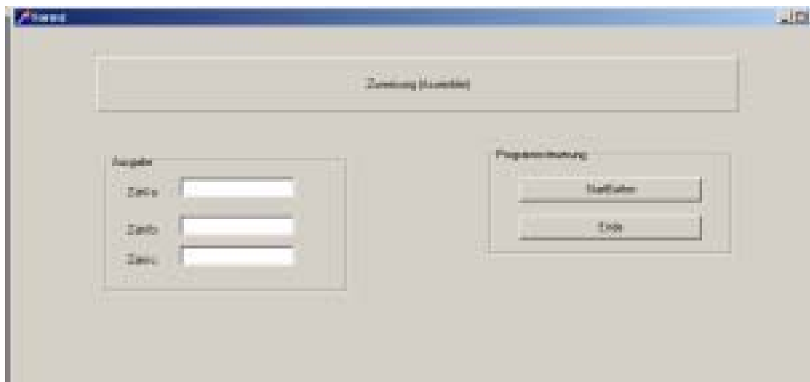
```
procedure wertzuweisung;
var a, b, c : word;
begin
        asm
                mov a, 20     { a ← 20 }
                mov b, 12     { b ← 12 }
                mov ax, a     { ax ← a }
                add ax, b     { ax ← ax+b }
                mov c, ax     {c ← ax }
                mov ax, a     { ax ← a }
                sub ax, b     {ax ← ax-b }
                mov a, ax     {a ← ax }
        end;
        Ausgabe(a,b,c)
end.
```

Assembler-Blöcke werden durch eine **asm**-Anweisung eingeleitet und mit **end** abgeschlossen.
Der Befehl **mov a, 20** ist ein Befehl mit Direktwertadressierung (Schiebe den Wert 20 in den Speicherplatz mit der Bezeichnung a, a ← 20).
Der Befehl **mov ax, a** ist ein Befehl mit direkter Adressierung (Schiebe den Wert im Speicherplatz a in den Akkumulator ax, ax ← a).
Der Befehl **add ax, b** addiert zum Akkumulator den Wert aus dem Speicherplatz b (ax ← ax + b).
Der Befehl **sub ax, b** subtrahiert vom Akkumulator den Wert aus dem Speicherplatz b (ax ← ax - b).

mov Ziel, Quelle Der mov-Befehl kopiert den Inhalt des Quellenoperanden in den Zieloperanden. Beide Operanden müssen vom gleichen Typ sein (byte/byte, word/word). Als Zieloperand kann jedes Register oder eine Speichervariable angegeben werden. Der Quelloperand kann auch aus einem Direktwert bestehen.

add Ziel, Quelle Der add-Befehl addiert die Inhalte des Ziel- und Quelloperanden und legt das Ergebnis im Zieloperanden ab.

sub Ziel, Quelle Der sub-Befehl subtrahiert den Inhalt des Quelloperanden vom Inhalt des Zieloperanden und speichert das Ergebnis im Zieloperanden ab.



```
unit UZuweisung;

interface

uses  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    GroupBox1: TGroupBox;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    aEdit: TEdit;
    bEdit: TEdit;
    cEdit: TEdit;
    GroupBox2: TGroupBox;
    StartButton: TButton;
    EndeButton: TButton;
    procedure StartButtonClick(Sender: TObject);
    procedure EndeButtonClick(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.StartButtonClick(Sender: TObject);

     procedure Ausgabe(a,b,c:word);
     begin
         aEdit.text:=IntToStr(a);
         bEdit.text:=IntToStr(b);
         cEdit.text:=IntToStr(c);
     end;

     procedure Wertzuweisung;
     var a,b,c:word;
     begin
         asm
           mov a,20          (* Eingabe a = 20 *)
           mov b,12          (* Eingabe b = 12 *)
           mov ax,a          (* Akkumulator  ax = a *)
           add ax,b          (* Addition  ax = ax + b *)
           mov c,ax          (* Speicherung c = ax = a + b *)
           mov ax,a          (* Akkumulator  ax = a *)
           sub ax,b          (* Subtraktion  ax = ax - b *)
           mov a,ax          (* Speicherung  c = ax = a - b *)
         end;
         Ausgabe(a,b,c);
     end;

begin  Wertzuweisung; end;

procedure TForm1.EndeButtonClick(Sender: TObject);
begin   close  end;

end.
```

4.2. Wertzuweisung (8-Bit-Multiplikation)

Der **mul-Befehl** multipliziert den Wert im Quelloperanden mit dem Wert im Akkumulator. Der Quelloperand kann ein CPU-Register oder eine Variable sein.
Die Byte-Multiplikation multipliziert zwei vorzeichenlose 8-Bit-Werte miteinander und legt das Ergebnis im AX-Register ab. Ein Operand muss sich dabei im AL-Register befinden. Der zweite Operand, der aus einem 8-Bit-Register oder einer Byte-Speicherstelle bestehen muss, wird beim mul-Befehl mit angegeben.

mul Quelle	Der mul-Befehl multipliziert den Quelloperanden mit dem AL-Register (Byte-Multiplikation) oder dem AX-Register bei einer Wort-Multiplikation

Multiplikand	*	Multiplikand	=	Ergebnis
AL (Byte)	*	Quelloperand (Byte)	=	AX (Wort)

```
procedure wertzuweisung;
var a, b : byte;
    c : word;
begin
        a := 20;
        b := 12;
        c := a * b;
        Ausgabe(a,b,c)
end.
```

```
procedure wertzuweisung;
var a, b : byte;
    c : word;
begin
        asm
                mov a, 20    { a ← 20 }
                mov b, 12    { b ← 12 }
                mov al, a    { al ← a }
                mul b        { ax ← al*b }
                mov c, ax    {c ← ax }
        end;
        Ausgabe(a,b,c)
end.
```

4.3. Wertzuweisung (8-Bit-Division)

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

div Quelle	Der div-Befehl dividiert das AX-Register durch den Quelloperanden (8-Bit-Divison)

Dividend	/	Divisor	=	Quotient	Rest
AX (16-Bit)	/	Operand (8-Bit)	=	AL	AH

```
procedure wertzuweisung;
var a : word;
    b, c, d : byte
begin
        a := 2000;
        b := 120;
        c := a div b;
        d := a mod b;
        Ausgabe(a,b,c,d)
end.
```

```
procedure wertzuweisung;
var  a : word;
     b, c, d : byte;
begin
        asm
                mov a, 2000  { a ← 2000 }
                mov b, 120   { b ← 120 }
                mov ax, a    { ax ← a }
                div b        { ax ← ax / b }
                mov c, al    { c ← al }
                mov d, ah    {c ← ah }
        end;
        Ausgabe(a,b,c,d)
end.
```

4.4. Einseitige Verzweigung

```
procedure einseitige_Verzweigung;
var a, b, c : word;
begin
        Eingabe(a,b);
        if a > b then begin
                        c := b;
                        b := a;
                        a := c
                      end;
        Ausgabe(a,b)
end;
```

```
procedure einseitige_Verzweigung;
var a, b, c : word;
begin
        Eingabe(a,b);
        asm
                        mov ax, a
                        cmp ax, b
                        jng  @marke1
                        mov ax, b
                        mov c, ax
                        mov ax, a
                        mov b, ax
                        mov ax, c
                        mov a, ax
                @marke1:
        end;
        Ausgabe(a,b)
end.
```

cmp Ziel, Quelle Der CMP-Befehl vergleicht zwei Operanden miteinander. Beide Operanden müssen gleich sein (Byte, Byte oder Word, Word). Das Ergebnis des Vergleichs wird im Statusregister angezeigt.
Der Vergleich erfolgt durch eine interne Subtraktion, d.h. Ziel - Quelle.
(CMP = CoMPare two operands)

Der Bezeichner eines lokalen Labels besteht aus einem Klammeraffen @ gefolgt von einem oder mehreren Buchstaben (A,.., Z, a,..,z), Ziffern (0,..,9), Unterstrichen (_) oder weiteren Klammeraffen(@).

Bedingte Sprunganweisungen:
Jede der Sprunganweisungen testet die Flags im Statusregister und führt einen Sprung nach Ziel aus, wenn die angegebene Bedingung erfüllt ist.

Zusammenstellung der wichtigsten Sprungbefehle

jmp	z	unbedingter Sprung nach Speicherplatz z	
jg	z	Sprung, falls Ergebnis > 0	(**G**reater than zero)
jng	z	Sprung, falls Ergebnis ≤ 0	(**N**ot **G**reater than zero)
je	z	Sprung, falls Ergebnis $= 0$	(**E**qual to zero)
jne	z	Sprung, falls Ergebnis $\neq 0$	(**N**ot **E**qual to zero)
jl	z	Sprung, falls Ergebnis < 0	(**L**ess than zero)
jnl	z	Sprung, falls Ergebnis ≥ 0	(**N**ot **L**ess than zero)

Befehl:	**Sprung wenn ....**	**geprüfte Statusflags:**
JA / JNBE	größer / nicht-kleiner-gleich above / not-below-equal	CF = 0 und ZF = 0
JAE / JNB	größer-gleich / nicht kleiner above-equal / not-above-equal	CF = 0
JB / JNAE	kleiner / nicht-größer-gleich below / not-above-equal	CF = 1
JBE / JNA	kleiner-gleich / nicht größer below-equal / not above	CF = 1 oder ZF = 1
JE / JZ	gleich / null equal / zero	ZF = 1
JG / JNLE	größer / nicht-kleiner-gleich greater / not-less-equal	ZF = 0
JGE / JNL	größer-gleich / nicht-kleiner greater-equal / not-less	SF gleich OF
JL / JNGE	kleiner / nicht-größer-gleich less / not-greater-equal	SF ungleich OF
JLE / JNG	kleiner-gleich / nicht größer less-equal / not-greater	ZF = 1
JNE / JNZ	nicht-gleich / nicht-null not-equal / not-zero	ZF = 0
JC	Carry Flag (CF) gesetzt Carry	CF = 1
JNC	Carry Flag (CF) nicht gesetzt not-Carry	CF = 0
JNP / JPO	keine Parität / Parität ungerade not-Parity / Parity-odd	PF = 0
JNO	kein Überlauf not-Overflow	OF = 0
JNS	kein Vorzeichen (positiv) not-Sign	SF = 0
JO	Überlauf Overflow	OF = 1
JP / JPE	Parität / Parität gerade Parity / Parity-even	PF = 1

Die Vertauschung der Variablen a, b kann durch den XCHG-Befehl vereinfacht werden.

xchg Quelle, Ziel XCHG vertauscht den Inhalt der Operanden. Beide Operanden müssen gleich groß sein, einer von ihnen muss ein Register sein. (eXCHanGe = Vertausche Inhalt der Operanden)

```
procedure einseitige_Verzweigung;
var a, b, c : word;
begin
        Eingabe(a,b);
        asm
                        mov ax, a
                        cmp ax, b
                        jng  @marke1
                        mov ax, b
                        xchg ax, a
                        mov b, ax
            @marke1:
        end;
        Ausgabe(a,b)
end.
```

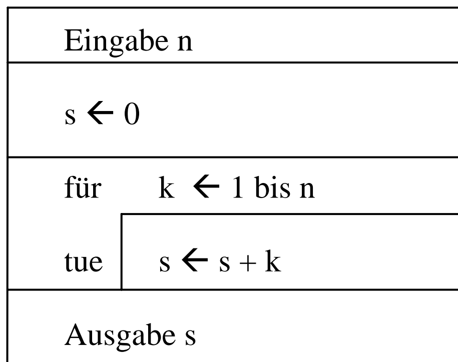
4.4. Zweiseitige Verzweigung

```
procedure zweiseitige_Verzweigung;
var a, b, c : word;
begin
        Eingabe(a,b);
        if a > b then c := a else c := b;
        Ausgabe(c);
end.
```

```
procedure zweiseitige_Verzweigung;
var a, b, c : word;
begin
        Eingabe(a,b);
        asm
        mov ax, a
                        cmp ax, b
                        jle    @marke1
                        mov ax, a
                        mov c, ax
                        jmp  @marke2
            @marke1: mov ax, b
                        mov c, ax
            @marke2:
        end;
        Ausgabe(c);
end.
```

jmp Ziel Der JMP-Befehl bewirkt einen Sprung auf eine Adresse im Programm, die mit dem Zieloperanden angegeben ist.
(JuMP = Unbedingter Sprung zur Zieladresse)

4.5. Gezählte Wiederholung (Summe der ersten n Zahlen)




```
procedure gezaehlte_Wiederholung;
var k, n, s : word;
begin
        Eingabe(n);
        s := 0;
        for k :=  1 to n do s := s + k;
        Ausgabe(s)
end;
```

```
procedure gezaehlte_Wiederholung;
var k, n, s : word;
begin
        Eingabe(n);
        asm
                        mov s, 0
                        mov k, 0
           @marke1:     mov ax, n
                        cmp ax, k
                        jng @marke2
                        inc k
                        mov ax, s
                        add ax, k
                        mov s, ax
                        jmp @marke1
           @marke2:
        end;
        Ausgabe(s)
end.
```

Die gezählte Wiederholung kann auch mit dem LOOP-Befehl formuliert werden.

LOOP Adresse Der LOOP-Befehl subtrahiert vom CX-Register den Wert 1 und überprüft anschließend das CX-Register auf null. Ist das CX-Register nicht null, wird zu der angegebenen Adresse verzweigt. Ist der Wert null, wird mit dem nächsten Befehl nach LOOP fortgeführt.
Das CX-Register wird mit dem Schleifenzähler geladen.
(LOOP until count complete = Springe zur Adresse, solange das CX-Register nicht null)

```
procedure gezaehlte_Wiederholung;
var s, k, n : word;
begin
        Eingabe(n);
         asm
                        mov s, 0
                        mov cx, n
            @marke1:    mov ax, s
                        add ax, cx
                        mov s, ax
                        loop @marke1;
        end;
        Ausgabe(s)
end.
```

4.6. Abweisende Wiederholung (Ganzzahliger Rest r = a mod b)

Eingabe a	
Eingabe b	
solange a $\geq$ b	
tue	a $\leftarrow$ a - b
r $\leftarrow$ a	
Ausgabe r	

```
procedure abweisende_Wiederholung;
var a, b, r : word;
begin
        Eingabe(a,b);
        while  a >= b do  a := a - b;
        r := a;
        Ausgabe(r)
end.
```

```
procedure abweisende_Wiederholung;
var a, b, r : word;
begin
        Eingabe(a,b);
        asm
            @marke1:    mov ax, a
                        cmp ax, b
                        jl      @marke2
                        mov ax, a
                        sub ax, b
                        mov a, ax
                        jmp @marke1
            @marke2:    mov ax, a
                        mov r, ax
        end;
        Ausgabe(r)
end.
```

4.7. Annehmende Wiederholung (Multiplikation p = a • b mit Hilfe der Addition)

<table>
<tr><td colspan="2">Eingabe a</td></tr>
<tr><td colspan="2">Eingabe b</td></tr>
<tr><td colspan="2">p ← 0</td></tr>
<tr><td rowspan="2">wiederhole</td><td>p ← p + a</td></tr>
<tr><td>b ← b - 1</td></tr>
<tr><td colspan="2">bis b = 0</td></tr>
<tr><td colspan="2">Ausgabe p</td></tr>
</table>

```
procedure abweisende_Wiederholung;
var a, b, p : word;
begin
        Eingabe(a,b);
        p := 0;
        repeat  p := p + a;
                b := b - 1
        until   b = 0;
        Ausgabe(p)
end.
```

```
procedure abweisende_Wiederholung;
var a, b, p : word;
begin
        Eingabe(a,b);
        asm
                        mov p, 0
              @marke1:  mov ax, p
                        add ax, a
                        mov p, ax
                        dec b
                        mov ax, b
                        cmp ax, 0
                        jne @marke1
        end;
        Ausgabe(p)
end.
```

5. Unterprogramme

a) Funktionen mit Werteparameter

```
function summe(a, b : word) : word;
begin
        summe := a + b
end;
```

```
function summe(a, b : word) : word;
begin
        asm
                mov ax, a
                add ax, b
                mov @result, ax
        end
end;
```

Mit @result wird das Funktionsergebnis zurückgeliefert und auf den Stack gelegt.

b) Funktionen mit Variablenparameter

```
function summe(var a, b : word) : word;
begin
        summe := a + b
end;
```

```
function summe(var a, b : word) : word;
begin
    asm
            les   bx, a
            mov ax, ES : [BX]
            les    bx, b
            add  ax, ES :  [BX]
            mov @result, ax
    end
end;
```

Der integrierte Assembler behandelt var-Parameter immer als 32-Bit-Zeiger mit einer Größe von vier Bytes. Möchte man auf den Inhalt eines var-Paramters zugreifen, muss zunächst der 32-Bit-Zeiger geladen werden, dann muss auf die Speicherstelle zugegriffen werden, auf die der Zeiger verweist.

LES Ziel, Quelle Der LES-Befehl überträgt die Segmentadresse des Quelloperanden in das Extrasegment ES und die Offsetadresse des Quelloperanden in den Zieloperanden.
(LES = Load ES-register)

6. Aufgaben

6.1. Übersetze jeweils die gegebene FUNCTION in ein Assembler-Programm.

6.1.1. ganzzahlige Division

```
function division(a, b : word) : word;
var x, y, q : word;
begin
        x := a;
        y := b;
        q := 0;
        repeat  x := x - y;
                q := q + 1
        until x < y;
        division := q
end;
```

6.1.2. Größter gemeinsamer Teiler ggt(a,b)

```
function ggt(a, b : word) : word;
var x, y : word;
begin
        x := a;
        y := b;
        while x <> y do if x > y then x := x - y
                                 else  y := y - x;
        ggt := x
end;
```

6.1.3. Quadratzahlen

```
function quadrat(n : word) : word;
var s, k : word;
begin
        s := 1;
        k := 1;
        while k <> n do begin
                                s := s + 2*k + 1;
                                k := k + 1
                        end;
        quadrat := s
end;
```

6.1.4. Summe der ersten Quadratzahlen $s = 1 + 4 + 9 + 16 + ... + n^2$

```
function summe_quadratzahlen(n : word) : word;
var summe : word;
begin
        summe := 0;
        for k := 1 to n do summe := summe + sqr(k);
        summe_qudratzahlen := summe
end.
```

7. Literatur

(1) Baumann, Hermes, Reimer
"Vom Programm zum Prozessor - Arbeitshefte Informatik"
Stuttgart 1994
Klett-Verlag
ISBN 3-12-717760-7

(2) Kaier
"Maschinennahe Programmierung unter MS-DOS"
Stuttgart 1991
Klett-Verlag
ISBN 3-12-717732-1

(3) "Turbo Pascal für Windows"
Programmierhandbuch
Starnberg 1991
Borland