Lehrgebiet

Assembler-programmierung

Görlitz, März 2018

Gliederung:

1. Einführung
2. Zyklus der Programmerstellung, debuggen
3. „Hello-World“ in Assembler
4. Macros für Ein- und Ausgabe im Console Mode
5. Prozessorstrukturen
6. Adressierung
7. Ausführbare Dateien im RM und PM
8. Grundelemente der Assemblersprache
9. Struktur eines Assemblerprogramms

10. Befehlssatz
11. Konsolen- Ein- und Ausgabe
12. Aufruf von BIOS- Routinen (nur RM)
13. Makro‘s, Includes, Unterprogramme, Bibliotheken
14. Verwendung des Inline-Assemblers in Visual Studio
15. Gleitkomma-Zahlen
16. Das erste Fenster-Programm
17. Der 64-bit Assembler

Ablauf der Lehrveranstaltung:

abwechselnd:

1. Woche	2h Vorlesung
2. Woche	2h Übung

Abschlußleistung:
120 min Klausur
(Programmieraufgabe am Computer)

1. Einführung

Vorteile:
- ergibt schnellsten und platzsparenden Code
- alle Prozessor-Ressourcen nutzbar (umfangreichster Befehlssatz im Vergleich mit anderen Sprachen)

Nachteile:
- genaue Kenntnis der Register und des Befehlssatzes notwendig
- lange Einarbeitungszeit notwendig
- lange Programmierzeit
- nur für einen CPU- Typ/eine -familie anwendbar (nicht portierbar)

Weiterentwicklung:

HLA – High Level Assembly

- von Randall Hyde
- erlaubt die Verwendung von Hochsprachen-Konstrukten in Assembler
- hilfreich für Anfänger, aber auch für Profis
- unterstützt erweiterte Datentypen und Objektorientierung
- Syntax ähnlich C++
- gut lesbarer Code
- erleichtert bei Kenntnis einer Hochsprache das Erlernen von Assembler

Anwendung:

- vorzugsweise Systemprogrammierung
  (Teile des Kernels)
  z.B. Umschalten vom Real Mode in Protected Mode
- bei reinen Anwendungsrechnern
  (in der Automatisierungstechnik)
- zeitkritische Anwendungen (Echtzeit)
- Lehre
  - Darstellung der Vorgänge im Rechner
  - Herstellung eines Bezugs zur Hardware

2. Zyklus der Programmerstellung

1. Editieren, Assemblieren, Linken, ausführbare Datei erstellen

- Visual Studio 2015
    - Verwendung des integrierten MASM32 (auch MASM64 möglich)

Beschreibung der Benutzung von VS 2015:
siehe Script „Bedienung MASMmitVS.doc“

evtl.:

2. Debuggen
   - „entwanzen“
   - Werkzeug zur Fehlersuche im Programm
   - Setzen von Schaltern bei Programmerstellung
   - Abarbeitung z.B. - im Schrittbetrieb
     - bis Breakpoint
   - Anzeige von - Registerwerten, Flags
     - Speicherinhalten (Hexdump)
     - Stackinhalten

3. „Hello-World“ in Assembler

```
1       .386
2       .model flat,stdcall
3       option casemap:none

4       include \masm32\include\windows.inc
5       include \masm32\include\masm32.inc
6       include \masm32\include\kernel32.inc

7       includelib \masm32\lib\kernel32.lib
8       includelib \masm32\lib\masm32.lib
```

```
9   .data
10      text db "Hallo Welt!",13,10,0

11  .code
12      main:
13      invoke StdOut, addr text
14      exit

15      end main
```

Erläuterungen zur Programm-Struktur:

1

- Angabe CPU (Registerbreite, Befehlssatz,...)

2

- Speichermodell (1 Segment, Offset 32 bit)

3

4- 8

- Einbindung von Quelltext- und Objektcode-Bibliotheken

9

- Beginn des Daten-Segments (kein eigenes Segment)

10

- Textlabel, einzelne Daten, Text, Position in Console (Spalte, Zeile)

11
- Beginn Code-Segments (kein eigenes Segment)

12
- Label, korrespondiert mit Label hinter *end*
- Festlegung des Eintrittspunktes (Startpunkt)

13
- Programmcode: Ausgabe von text über die Windows-Standard- Ausgabe StdOut

14
- Aufruf der exit- Funktion

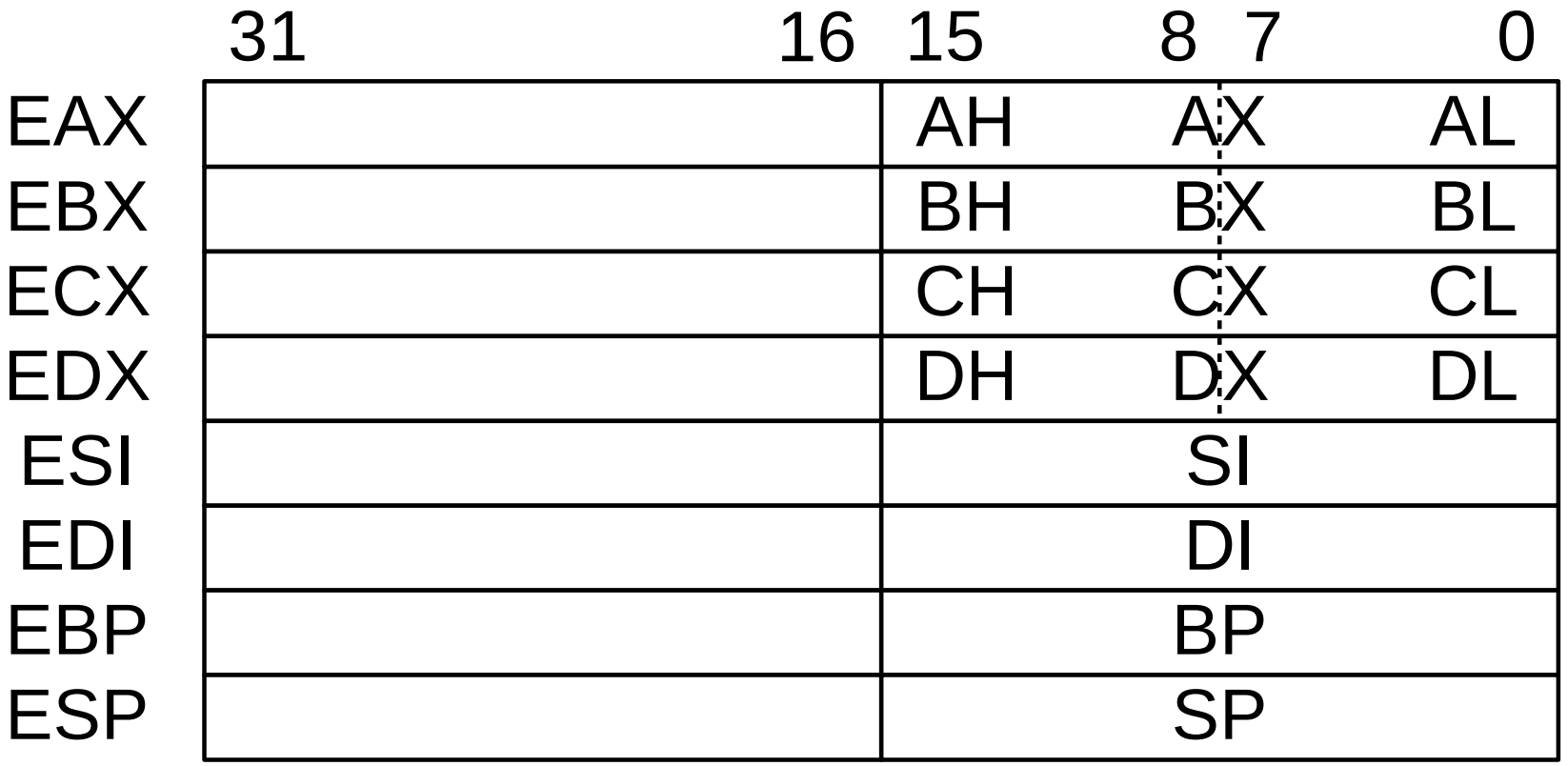
15
- logisches Programmende (mit Label für Einsprung)

5. Prozessorstrukturen

5.1. x86-Generationen

Generation	z.B. CPU
1	8086
1 update	80186
2	80286
3	80386
4	80486
5	Pentium
6	Pentium Pro, Pentium II, Pentium III
7	Pentium 4
8	Xeon, Core 2
9	Core i7(ab Mitte 2015 6.Generation)

5.2. 80386
5.2.1. Register
5.2.1.1. Datenregister



- EAX, EBX, ECX, ..
    - 32 Bit-Register
- AX, BX, CX, …
    -16 Bit-Register
- AH, AL, BH, BL, …
    - 8 Bit-Register
- andere Teile sind NICHT ansprechbar

maximaler Wertebereich:

8-Bit:	unsigned	0…255
	signed	-128…127
16-Bit:	unsigned	0…65535
	signed	-32.768…32767
32-Bit:	unsigned	0…4.294.967.295
	signed	-2.147.483.648…2.147.483.647

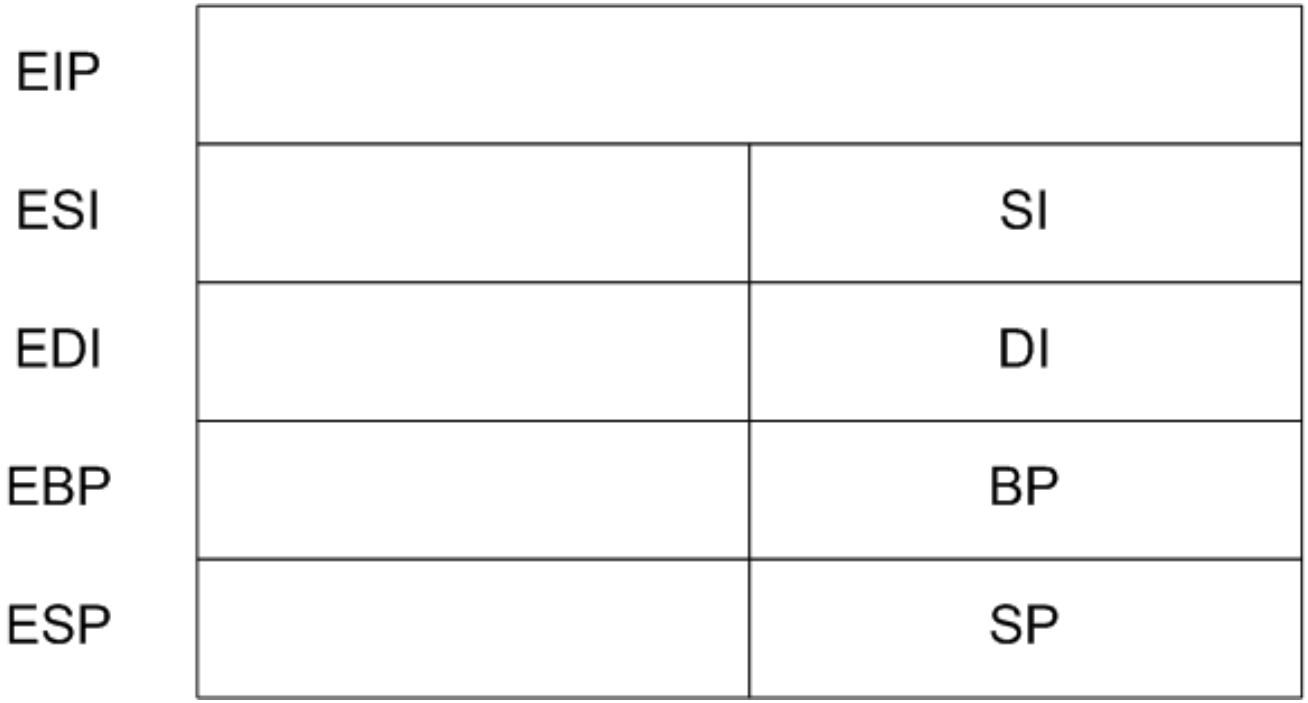
-alle Datenregister als temporäre Speicher nutzbar
- spezielle Nutzung:

EAX
- Akkumulator/Ergebnisspeicher für 32-Bit-Ops
- verwendet bei 32-Bit Multiplikation und Division
- Datenregister für E/A-Operationen AX
- Akkumulator/Ergebnisspeicher für 16 Bit-Ops
- verwendet bei 16-Bit Multiplikation und Division
- Datenregister für E/A-Operationen AL
- Akkumulator/Ergebnisspeicher für 8-Bit-Ops
- verwendet bei 8-Bit Multiplikation und Division

EBX - Basisregister
- für Basis-Adressierung im Speicher
ECX - Zählregister
- Iterationsregister bei Schleifenbefehle
- Elementzähler bei sich wiederholenden Stringoperationen
CL
- Zählregister bei Bit-Schiebe- und Rotationsbefehlen
EDX
- Akku-Erweiterung bei 32 Bit-Multiplikation und 32 Bit-Division
DX
- Akku-Erweiterung bei 16 Bit-Multiplikation und 16 Bit-Division

5.2.1.2. Pointerregister

- Zeiger für spezielle Aufgaben
- NICHT als Datenregister nutzbar



EIP
- Befehlszähler
- zeigt auf das 1.Byte (Op.-code) des nächsten Befehls
- Offset in Bezug zum Codesegment-Anfang (CS)

ESP - Stapelzeiger
- Offset in Bezug zum Stackanfang (SS)
- enthält Offset des niederwertigen Datenbytes des zuletzt im Stack abgelegten Wertes

Stack- Kellerspeicher
- Ablage für Rücksprungadressen, Register,…
- ESP wird vor Benutzung des Stacks dekrementiert

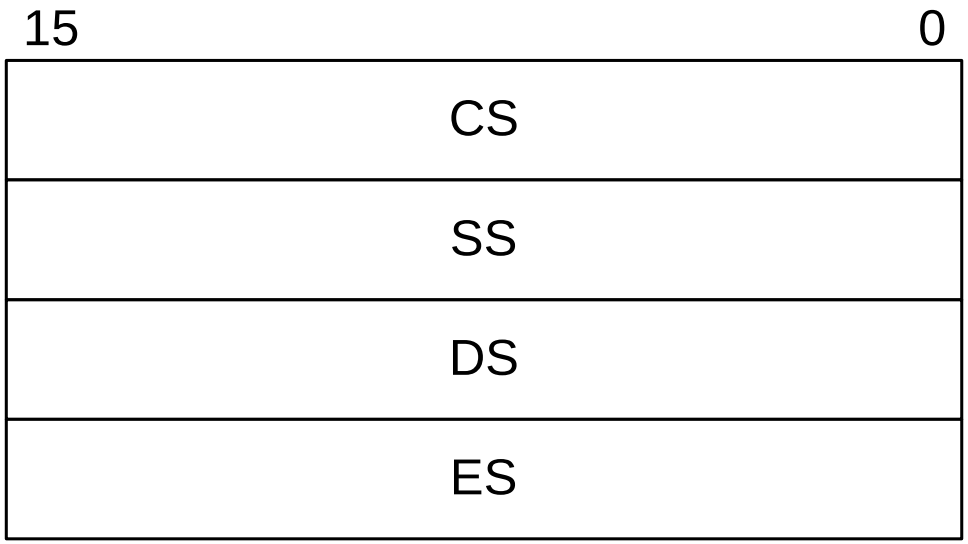
ESI,EDI

- enthalten bei Stringoperationen die Offset-adressen für das Datensegment (ESI) und das Extrasegment (EDI)

EBP

-Zeiger für Adressierung im Stack

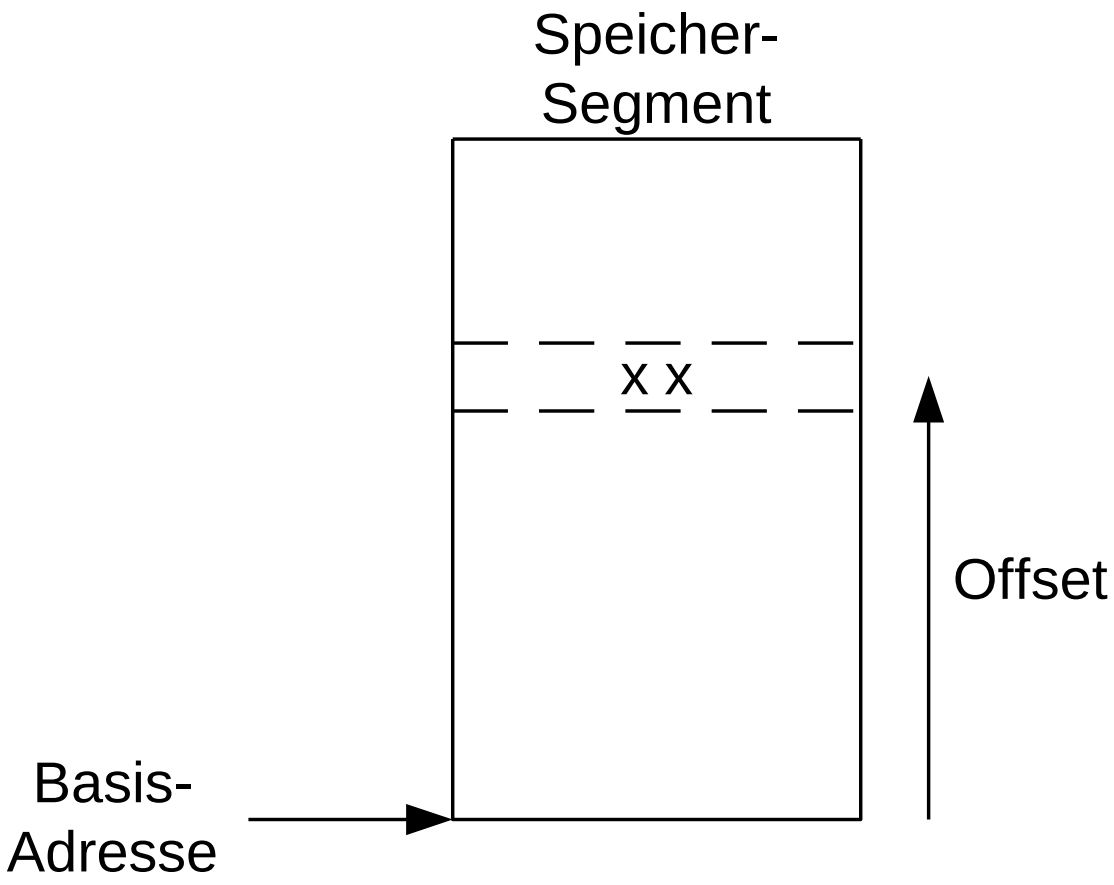
5.2.1.3. Segmentregister



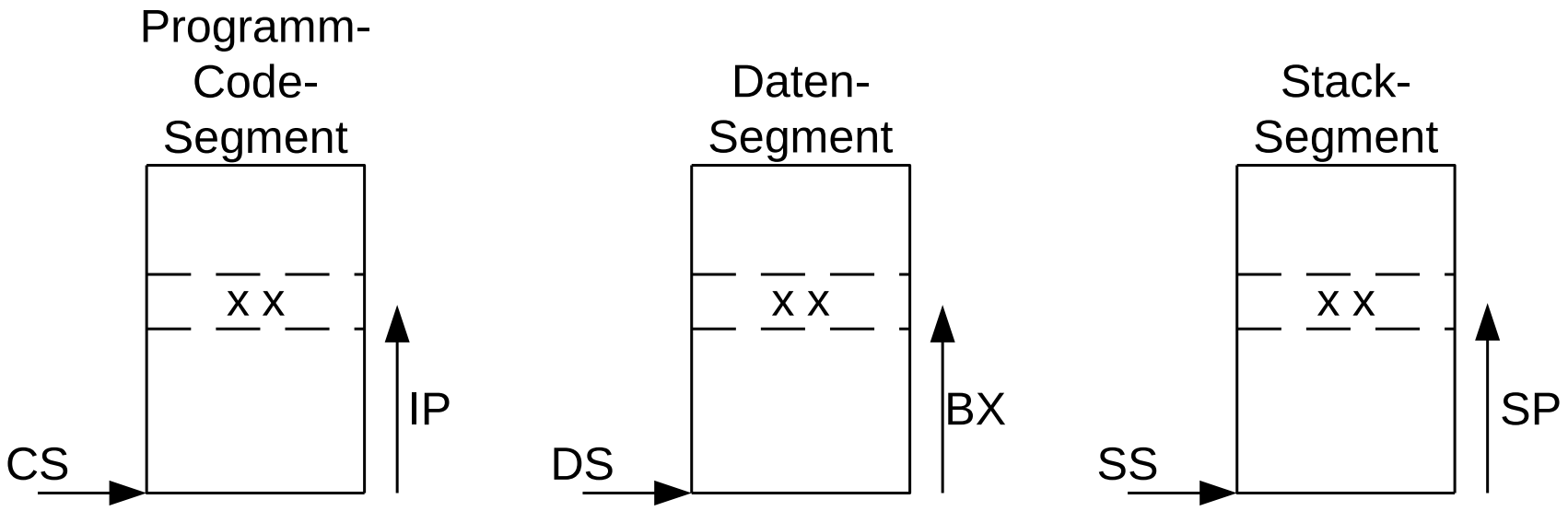
Verwendung:

- Befehle und Daten werden nicht in demselben Speicherbereich abgelegt
- Speicherbereich ist in Blöcke von eingeteilt; bei Modell „flat“ nur 1 Speicherbereich, der in Blöcke eingeteilt wird
- Verwaltung durch Segmente um Code von Daten und Stack zu trennen
- Basisadressen dieser Segmente stehen in
  → Segmentregistern
  CS → enthält Basisadresse des Codesegments
  SS → enthält Basisadresse des Stacksegments
  DS → enthält Basisadresse des Datensegments
  ES → enthält Basisadresse des Extrasegments
  FS, GS →Basisadresse für weitere Datensegm.

Bildung einer Adresse:



konkret für die einzelnen Segmenttypen:

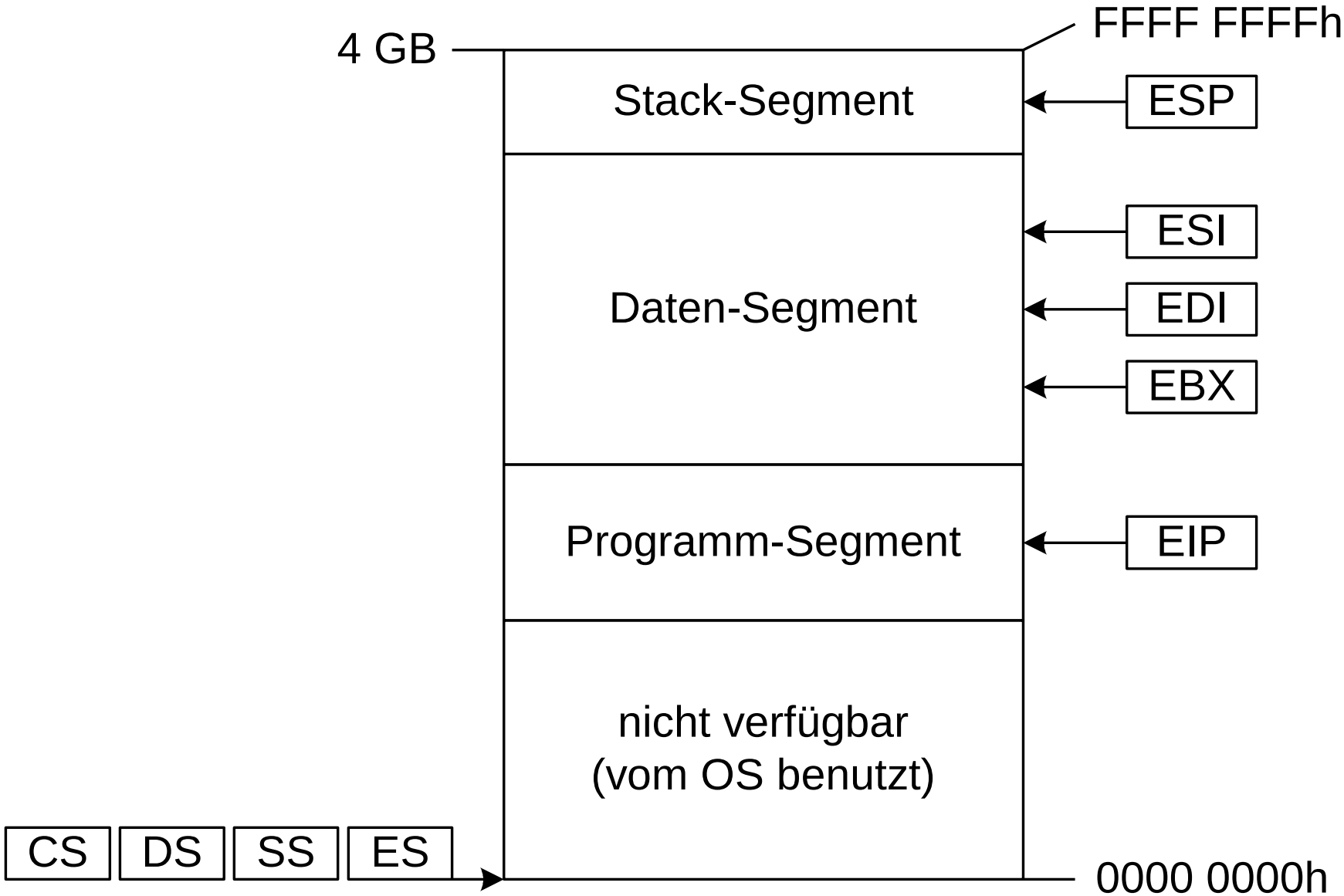


RM:

- Segmente sind einheitlich 64 KByte groß (wegen 16-bit IP)
- bei größerem Code oder größeren Daten → Verteilung über mehrere Segmente

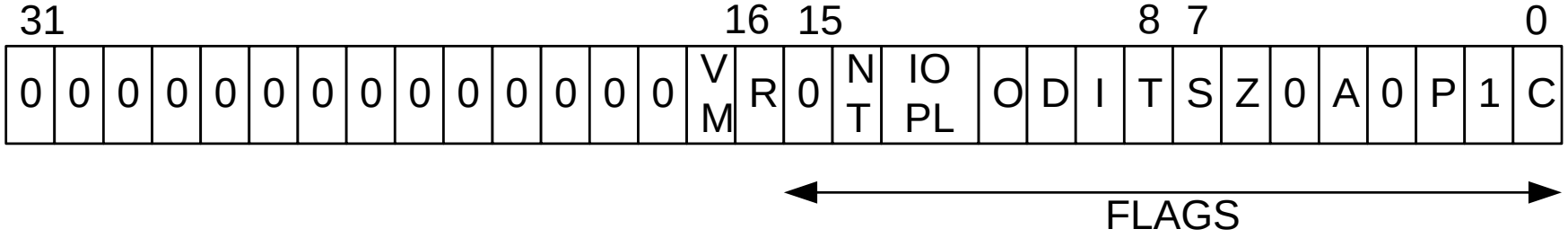
PM:

- Segment-Inhalte haben andere Aufgaben: Sie bestimmen (indirekt) die Anfangsadresse des 4GByte-Flat-Segments im virtuellen 4GByte Speicher
- Segment-Anfangsadressen können nicht vom Nutzer verändert werden, Sie sind geschützt vom Betriebssystem



Speicheraufbau im Flat-Modell

5.1.1.4. Programmstatuswort (PSW) beim 80386



- enthält Flag-Bits
- zeigen die Art des Ergebnisses von z.B. logischen oder arithmetischen Operationen an

C - Carry (Übertrag)

= 1 - wenn bei Addition ein Übertrag entsteht
bei Subtraktion ein Borgen entsteht

- wird bei einigen Schiebe- und Rotations-befehlen mit einbezogen
- zur Auswertung von log. oder arithm. Operationen

P – Parity (Parität)
= 1 - wenn das Ergebnis der vorherigen Operation eine gerade Anzahl von „1“ enthält
- Verwendung bei Datenübertragungen (z.B. asynchrone serielle Datenübertragung)

A – Auxiliary Carry (Hilfsübertrag)
= 1 - wenn bei BCD-Operationen ein Übertrag (Addition/Subtraktion) des Operanden auftritt (Übertrag von Bit D3 nach D4)

Z – Zero (Null)
= 1 - wenn Ergebnis Null ist
- zur Auswertung von Vergleichsoperationen

O – Overflow (Überlauf)

= 1 - wenn Addition von 2 vorzeichenbehafteten Operanden mit gleichem Vorzeichen oder Subtraktion von 2 vorzeichenbehafteten Zahlen mit unterschiedlichen Vorzeichen ein Resultat liefert, das den Wertebereich des Zweierkomplements überschreitet

- wenn sich das höchstwertigste Bit einer vorzeichenbehafteten Zahl während einer (arith.) Schiebeoperation verändert
- wenn das Ergebnis einer Division die Kapazität des Zielregisters überschreitet

S – Sign (Vorzeichen)
= 1 - wenn Ergebnis einer Operation mit vorzeichenbehafteten Zahlen negativ ist
- stellt das höchste Bit einer 8-Bit- oder 16-Bit-Operation dar

Bemerkung:
Auswertung von Multiplikationsergebnissen:
C = 1 und O = 1 → höherwertiges Byte bzw. Wort ≠ 0
C = 0 und O = 0 → höherwertiges Byte bzw. Wort = 0

Kontrollflags

T – Trap (Einzelschritt)
= 1 - CPU wird in Einzelschrittbetrieb gesetzt
(nur im Stack manipulierbar)

I – Interrupt (Unterbrechung)
= 1 - Freigabe maskierbarer Interrupts

D – Direction (Richtung)
= 1 - bei Stringoperationen wird das Indexregister abwärts gezählt
= 0 - bei Stringoperationen wird das Indexregister aufwärts gezählt

IOPL - I/O-Privileg Level (2 Bits)
- Privilegierungsstufe bei Ein-Ausgabebefehlen
- 0 ist höchste Privilegierungsstufe

NT - Nested Task Flag
-zur Anzeige von Taskverschachtelungen

R
- zur Steuerung, ob Exceptions verhindert werden sollen

VM
- zur Aktivierung des virtual 8086-Modes

5.3. 64-bit-CPU (Intel)

allgemeine Register (64-bit):
RAX, RBX, RCX, RDX

Zeiger-Register (64-bit):
RBP, RSI, RDI, RSP, RIP

Registererweiterung (64-bit):
R8, R9, …, R15

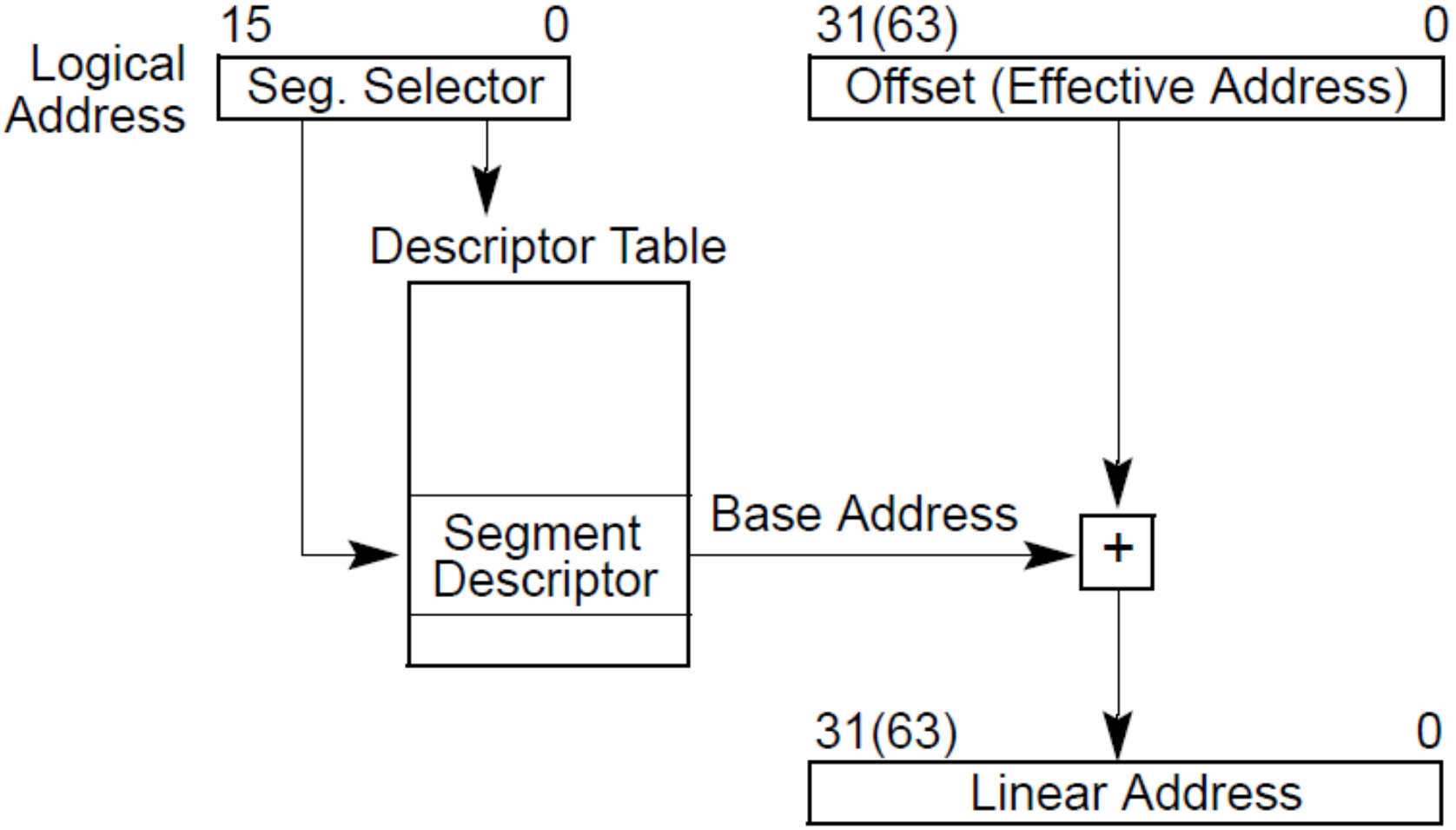
Multimedia-Erweiterung (64-bit):
MM0/ST0, MM1/ST1,…, MM7/ST7

Flag-Register (32-bit):
EFLAGS

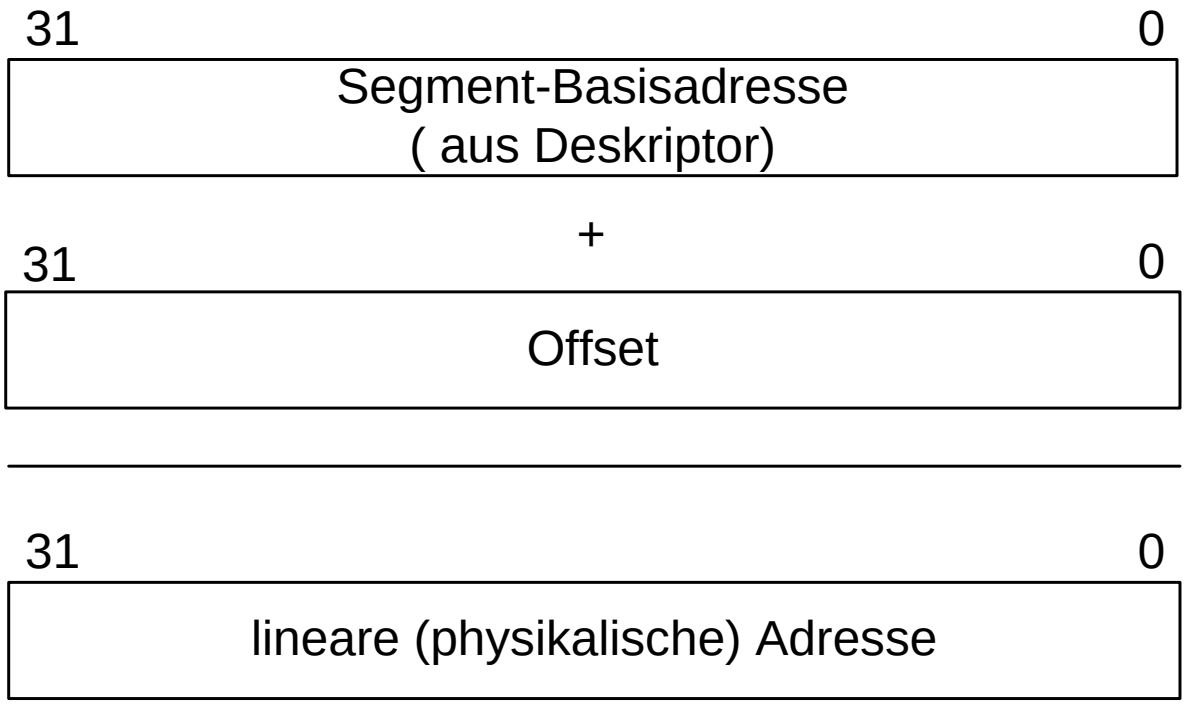
Streaming SIMD- Erweiterungs-Register (64-bit):
XMM0, XMM1,…,XMM15

6. Adressierung

6.1. Logische und lineare Adresse /1, 3-4/

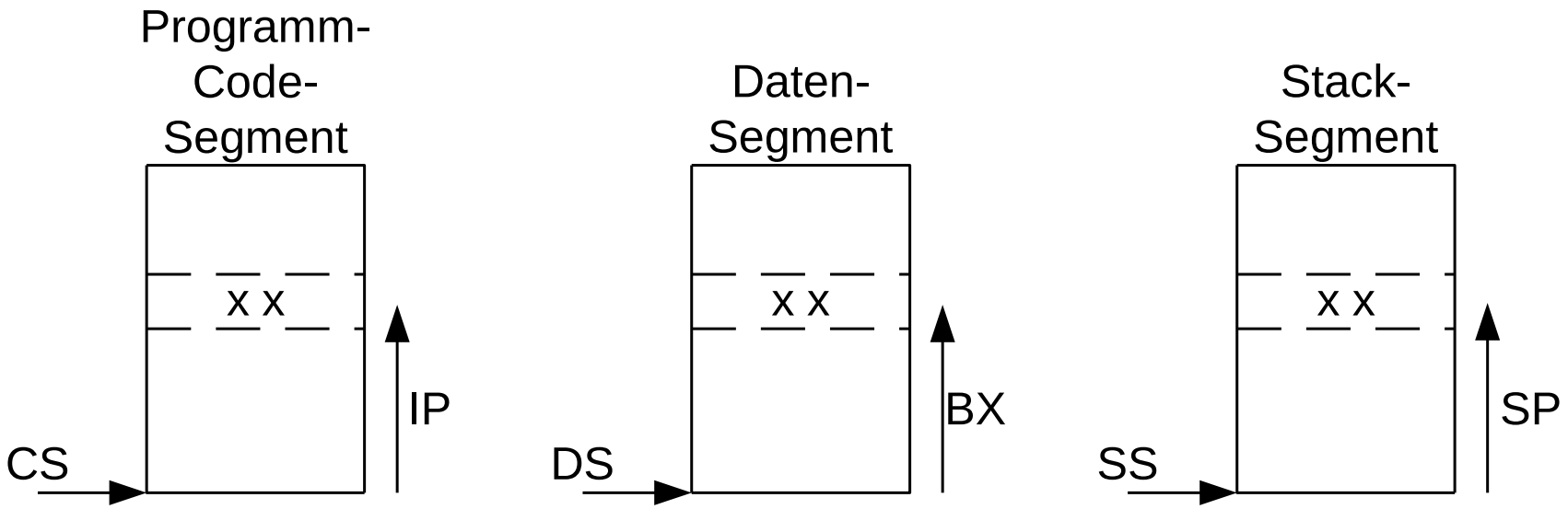


Bildung der linearen Adresse

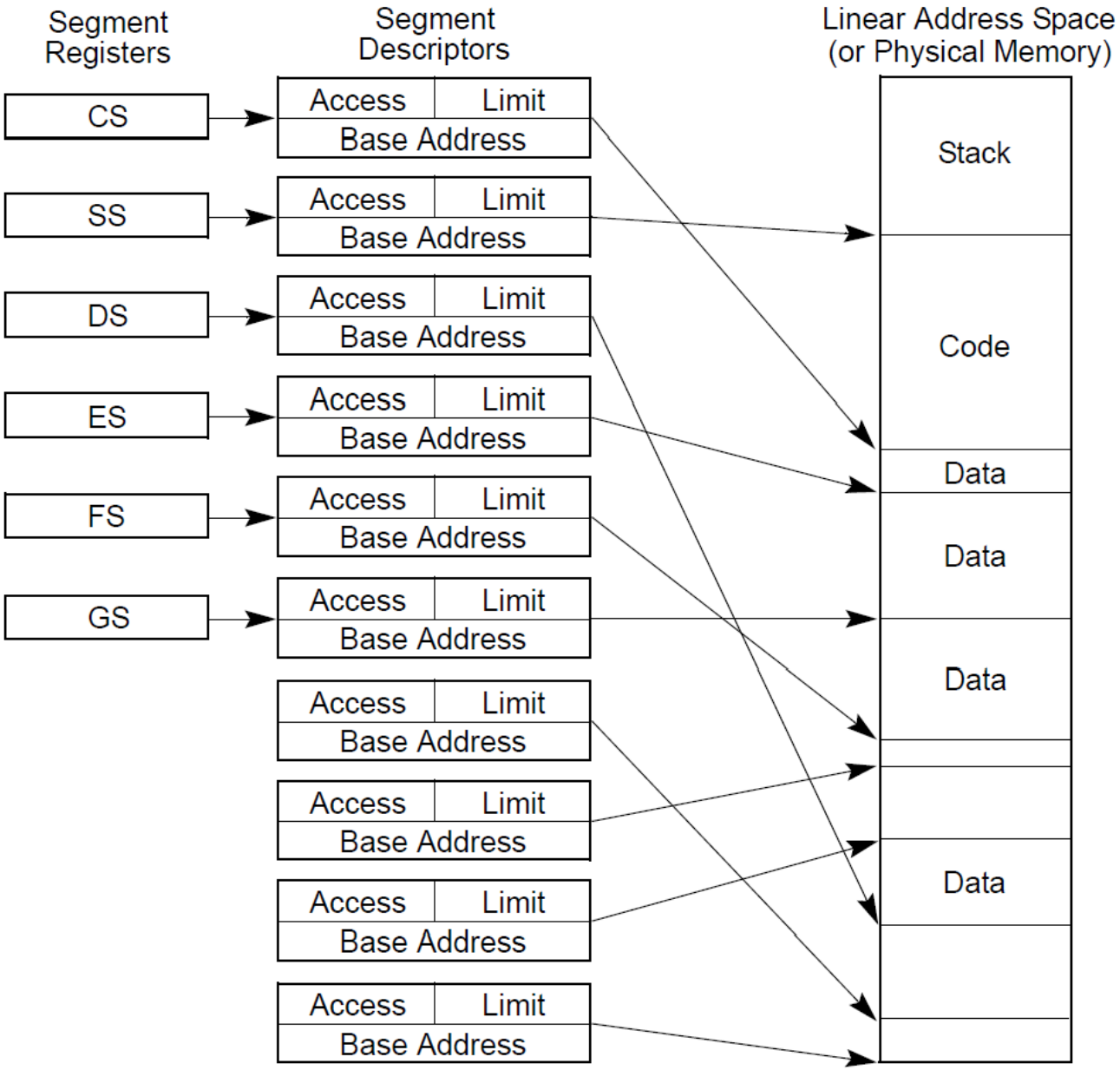


- die Basisadresse wird dem Deskriptor entnommen
- die Offsetadresse wird im Befehl angegeben

im konkreten Fall ist der Offset:



das Multi-Segment-Modell /1, 3-6/



6.2.2. Adressierungsarten

- Art und Weise, wie Adresse der Operanden - (Register und / oder Speicherplätze) angegeben wird
- in den Beispiele wird nur mit 32-Bit-Registern gearbeitet

→ nicht alle Befehle erlauben alle Adressierungs-Arten (siehe Intel-Befehlsliste)

Operanden:

können
- im Befehl direkt angegeben werden
- sich in Registern befinden
- im Speicher befinden
- auf E/A-Ports befinden

- Adressierung eines Operanden durch
  Basis : Offset
- allgemeine Zuordnung

Programmcode	→ Codesegment
Daten	→ Datensegment
Stackdaten	→ Stacksegment

Änderung der vorgegebenen Zuordnung ist in einigen Fällen möglich

→ Segment-Override-Präfix

Ausnahmen:
- bei Zeichenkettenoperationen muss die Zieladresse immer über das Segmentregister ES angegeben werden
- Operanden, die mit Hilfe des Stackpointers ESP adressiert werden, sind nur über das Stacksegment-register SS erreichbar

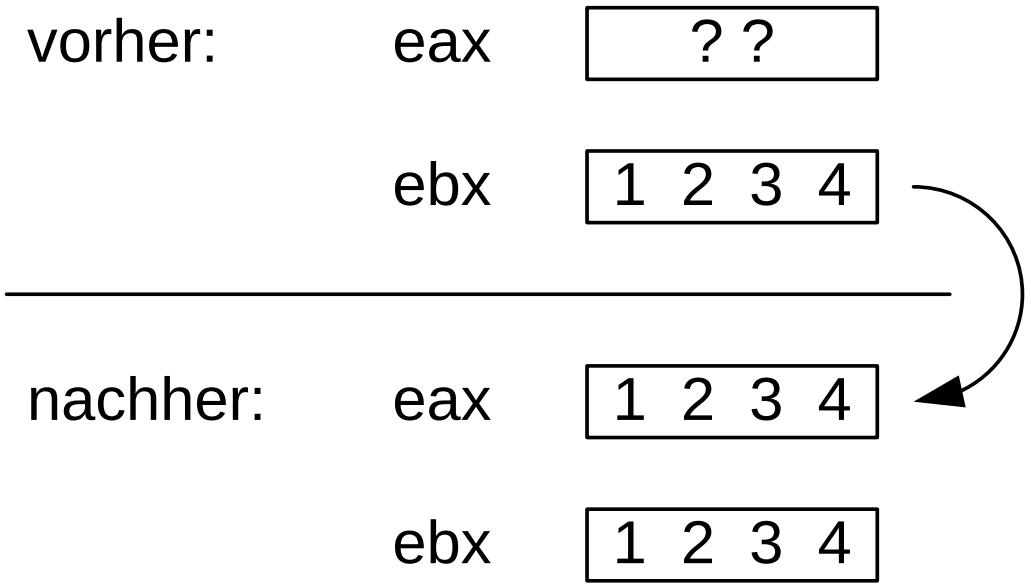
Offset kann aus einer oder mehreren Komponenten gebildet werden:
- Distanzwert im Befehl
- Offset ist Inhalt eines der beiden Register EBX, EBP
- Offset ist Inhalt eines der beiden Register ESI, EDI

jede Komponente kann negativ oder positiv sein !

1. Register-direkte Adressierung (implizite Adressierung)

Operand steht in einem Daten-, Basis- oder Index-register

Beispiel: mov eax, ebx



2. Unmittelbare Adressierung

8-Bit-,16-Bit- oder 32-Bit- Konstante wird direkt in ein Register geladen

Beispiel:

```
mov eax, 200
mov ebx, -10
mov ecx, 10001000b
mov edx, 88h
```

3. Direkte Adressierung

- Operandenadresse steht
  - als Speicheradresse
  - als Label (Marke)
- verwendet das Datensegmentregister DS; ein Override ist möglich

Beispiel: …

```
.data
tabelle     db     10h, 20h, 30h
.code
        mov al, [tabelle]
…
nachher: al = 10h
```

4. Register-indirekte Adressierung

Operandenadresse steht

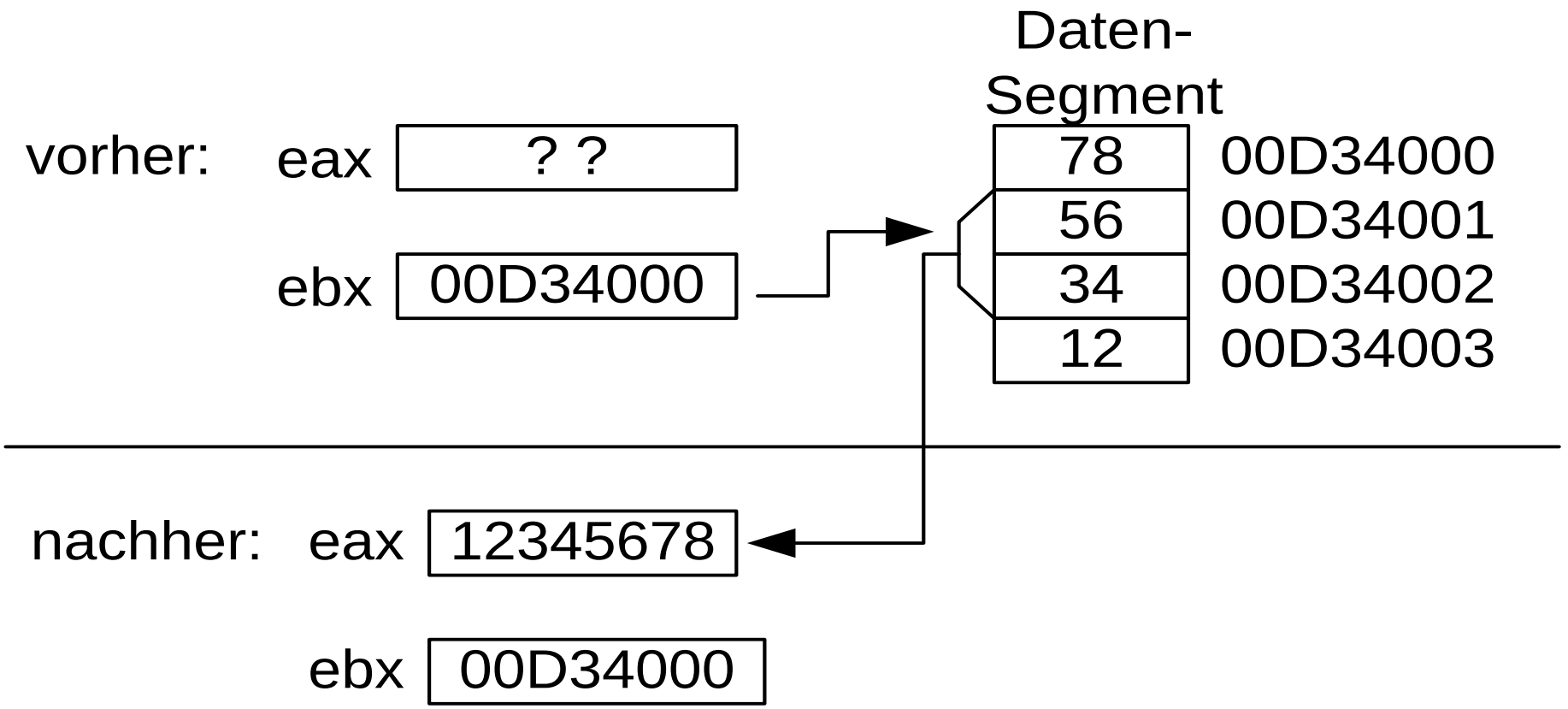
- im Basisregister EBX
- im Zeigerregister EBP
- im Indexregister ESI oder EDI

Operand muss in „[ ]“ stehen

Beispiel:

```
.data
        var1 dd 12345678h
.code
        mov ebx, offset var1
        mov eax, [ebx]
```

Intelkonvention:
niederwertiges Byte auf niederwertiger Adresse
höherwertiges Byte auf höherwertiger Adresse
Annahme: Adresse von var1 ist 00D34000h



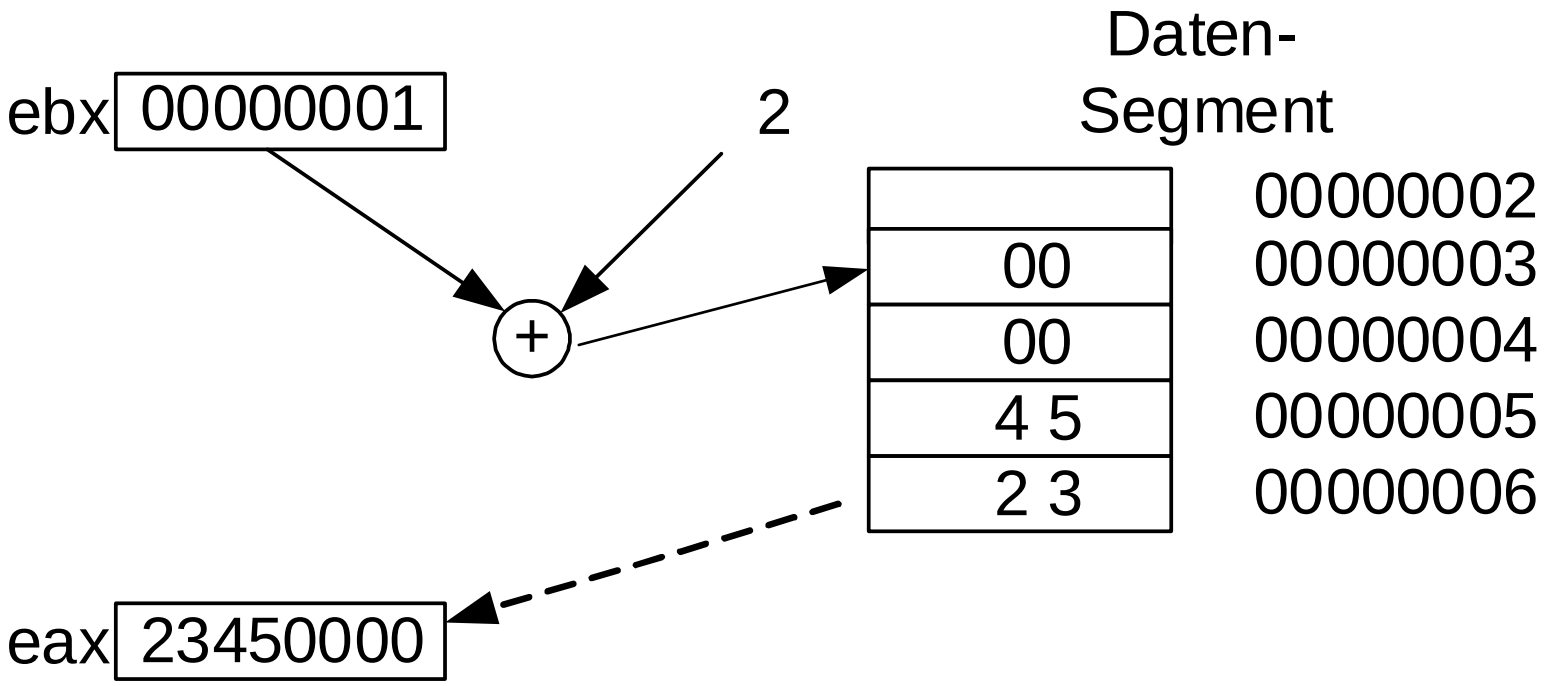
5. Basis-relative Adressierung

Offsetadresse:

Inhalt EBX oder EBP

+ Distanzwert im Befehl

Beispiel: mov eax, [ebx +2]



6. Direkt indizierte Adressierung

Offsetadresse:

Distanzwert im Befehl

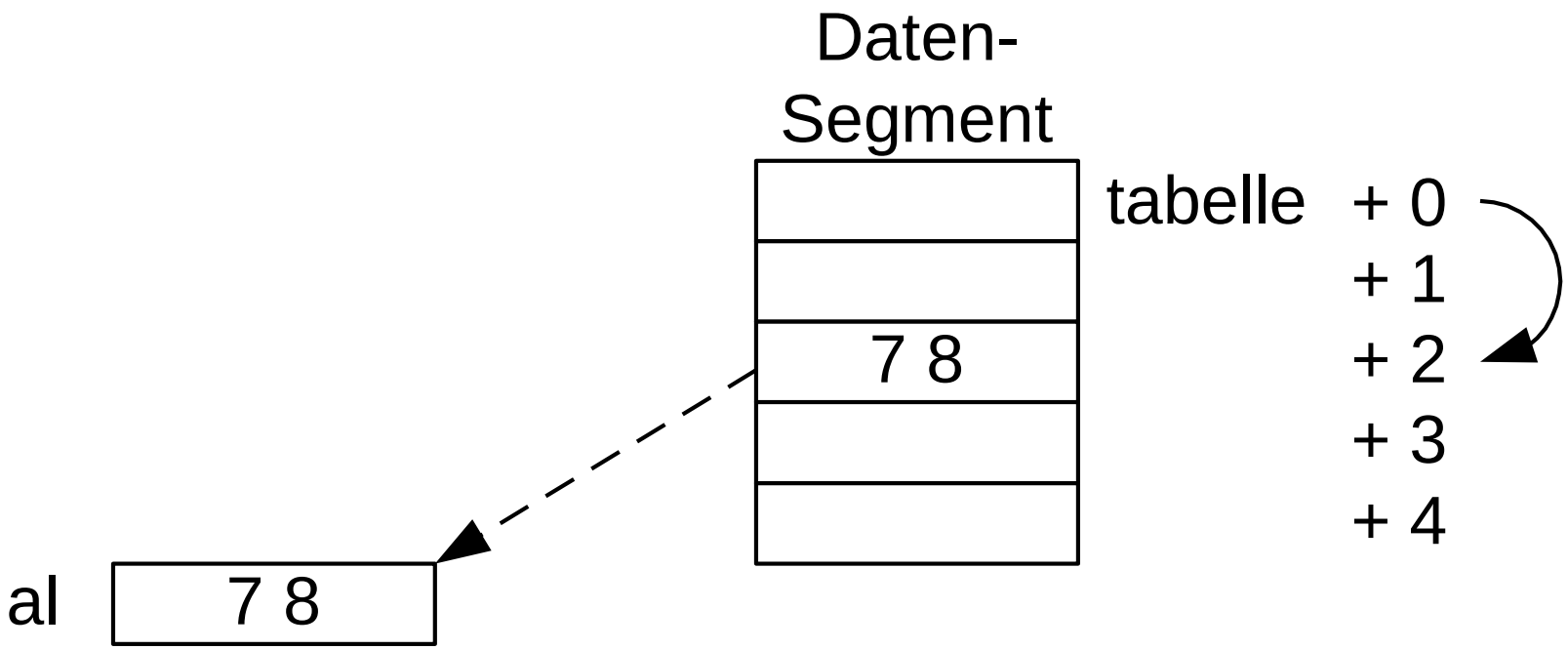
+ Inhalt ESI oder EDI

- Distanzwert gibt Startadresse des Feldes an
- ESI / EDI gibt die Entfernung an zur Startadresse (Offset)

Anwendung: Zugriff auf Tabellen-Elemente
Tabelle ist byte-weise organisiert

Beispiel:

```
mov edi, 2
mov al, tabelle[edi]
```


Daten-
Segment
tabelle
+ 0
+ 1
+ 2
+ 3
+ 4
7 8
al
7 8

7. Basis-indizierte Adressierung und (optionale) Verschiebung

Offsetadresse:

Inhalt EBX oder EBP

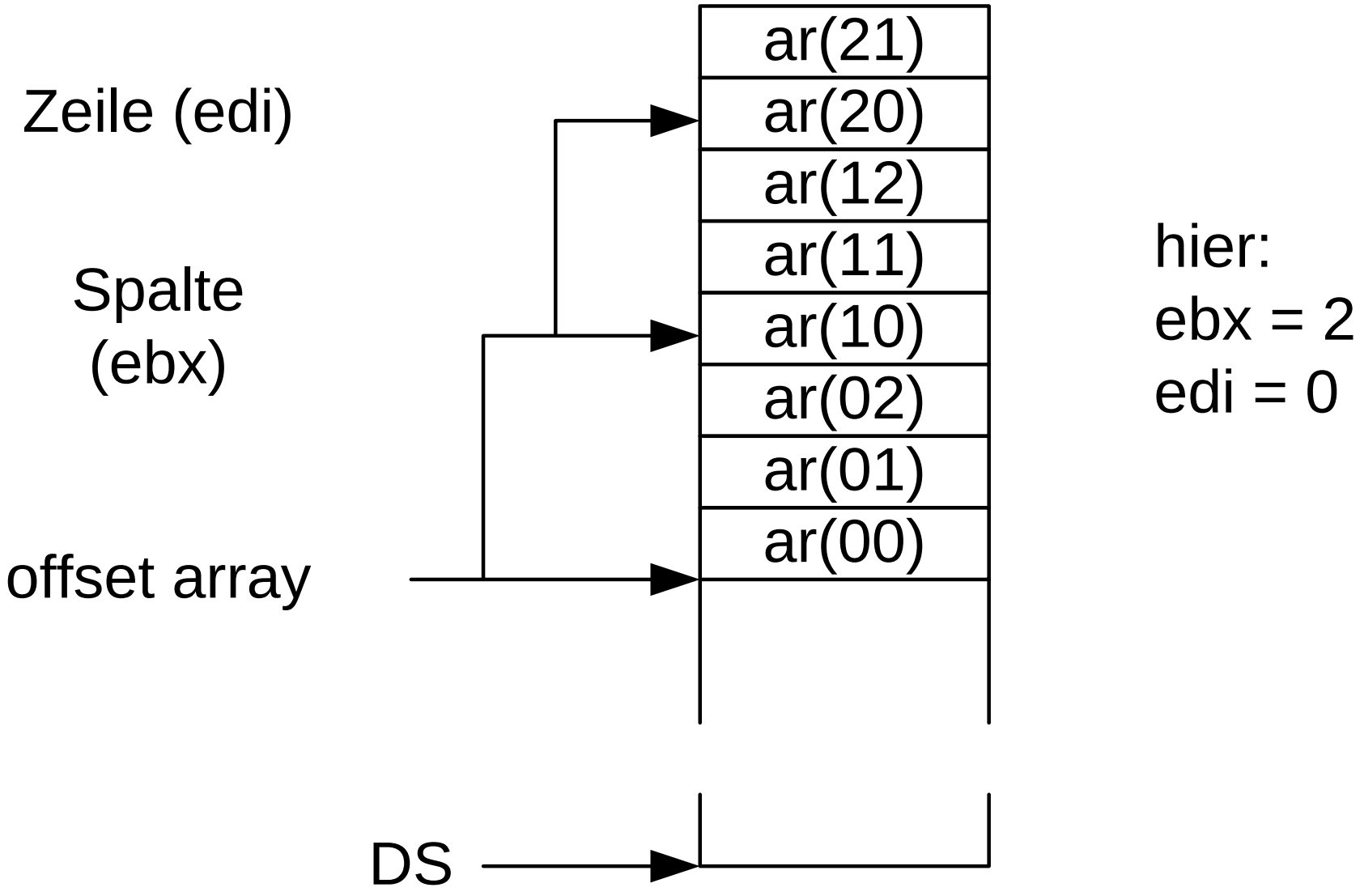
+ Inhalt Indexregister ESI oder EDI

(+ Distanzwert im Befehl) ← optional

- Distanzwert ist Startadresse des Feldes
- z.B. Basisregister → Spalte
  Indexregister → Zeile

Anwendung: Zugriff auf 2-dimensionale Felder, byte-weise organisiert

Beispiel: mov al, array[ebx][edi]


Zeile (edi)
Spalte
(ebx)
offset array
DS
ar(21)
ar(20)
ar(12)
ar(11)
ar(10)
ar(02)
ar(01)
ar(00)
hier:
ebx = 2
edi = 0

8. Stringadressierung

beide Operanden können vom Typ Speicher sein

Quelladresse:
- Offset ESI
- Segmentadresse DS (Override möglich)

Zieladresse:
- Offset EDI
- Segmentadresse ES

7. Ausführbare Dateien im RM und PM

- „exe“ – Datei-Endung für ausführbare Dateien
- häufigste Arten
- Programme für MS-DOS
- Programme für MS Windows
- beide beginnen mit dem Header einer MZ-Datei
- Formate:
    - Dateien für 16-Bit-Windows: NE-Format (New Executable), auch unter 32-Bit Windows ausführbar
    - Dateien für 32-Bit und 64-Bit-Windows: PE-Format (Portable Executable); Format ausführbarer Dateien in EFI-Umgebungen

8. Grundelemente der Assemblersprache

8.1. Grundlemente

1. Integer- Konstanten

[(+ | -)] digits [radix]

radix: h hexadezimal

o octal

d dezimal

b binär

r encoded real

ohne radix: dezimal

2. Reelle Konstanten

[sign] integer. [integer] [exponent]
sign {+, -}
exponent E[{+, -}] integer

Beispiele: 1.
+4.2
-23.6E+07
67.E2

3. Zeichen- Konstanten

- einzelnes Zeichen in einfachen oder doppelten Hochkommatas
  - ‘A‘
  - “a“
- MASM speichert Zeichen im Speicher als ASCII-Code

4. Zeichenketten- Konstanten

- mehrere Zeichen, einschließlich des Leerzeichens in einfachen oder doppelten Hockommata
  - ‘ABCD‘
  - “Hello World!“

5. Schlüsselworte

- sind reservierte Worte
- können nur in deren korrektem Kontext verwendet werden
- Befehle, wie „mov“, „add“
- Register- Namen
- Direktiven
- Attribute wie „byte“ , „word“ oder „dword“
- Operatoren in konstanten Ausdrücken
- vordefinierte Symbole, wie „@data“

6. Bezeichner

- vom Programmierer vergebener Name
- bezeichnet werden: Variablen
  Konstanten
  Prozeduren
  Label (Marken)
- darf 1 bis 247 Zeichen lang sein
- nicht case-sensitiv
- 1. Buchstabe muss sein: Buchstabe
  _, @, ?, §
- darf kein reserviertes Schlüsselwort sein

7. Direktiven

- definieren Variablen, Makros, Prozeduren
- nicht case-sensitiv

```
var1 dword 1234
```

- wichtige Direktiven → Segment-Definitionen

```
.stack
.data
.code
```

- ein neue Segment-Definition schließt automatisch das vorhergehende Segment

8. Befehle

- Anweisungen, die von der CPU ausgeführt werden nach der Übersetzung in die Maschinensprache
- ein Befehl kann folgende Teile enthalten:
    - label (optional)
    - Befehlscode
    - Operanden (0 | 1 | 2)
    - Kommentare (optional)

Label

- Bezeichner zur Kennzeichnung, an welcher Speicheradresse der Befehl oder das Datum steht

Daten- Label: var1 dword 200

Code- Label: m1: mov, ax, bx

Kommentare
- erhöhen die Lesbarkeit des Codes
- können z.B. enthalten
- Namen der Programmierer
- Version und Datum
- Bemerkungen über die Implementierung

Einzeilen- Kommentar
; das ist ein Einzeilen- Kommentar
Mehrzeilen-Kommentar
COMMENT !
Kommentarzeile 1
Kommentarzeile 2
!

8.2. Daten- Definitionen

[name] directive initializer [, initializer]

mögliche Direktiven

byte	8-bit Integer, ohne Vorzeichen
sbyte	8-bit Integer, mit VZ
word	16-bit Integer, ohne VZ
sword	16-bit Integer, mit VZ
dword	32-bit Integer, ohne VZ
sdword	32-bit Integer, mit VZ
fword	48-bit Integer (far pointer)
qword	64-bit Integer
tbyte	80-bit Integer
real4	32-bit short Real
real8	64-bit long Real
real10	80-bit extended Real

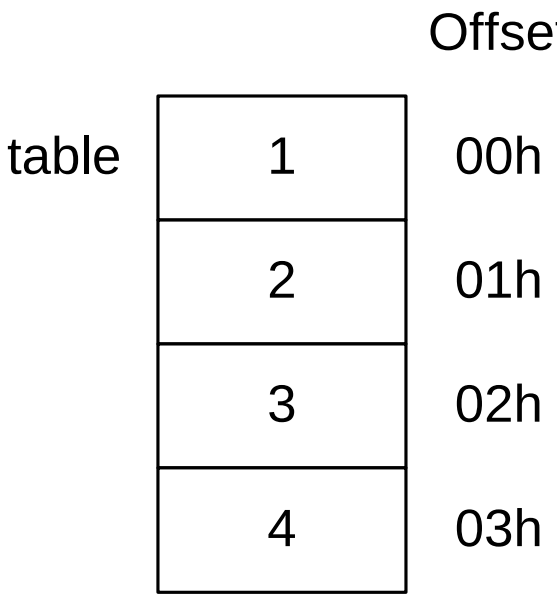
auch mögliche Direktiven

db	8-bit Integer
dw	16-bit Integer
dd	32-bit Integer
dq	64-bit Integer
dt	80-bit Integer

```
Beispiele:      val1    byte    200
                val2    db      200
                val3    sbyte   -14
          v     al4     db      -14
```

Mehrfache Initialisierung

table db 1,2,3,4



- auch mit verschiedenen Radixen erlaubt

table db 10, 45, 13h, 01011010b

Definition von Zeichenketten

```
message1  db     “hello world!“
message2  db     ‘h‘,‘e‘,‘l‘,‘l‘,‘o‘, …usw.
```

In vielen Programmiersprachen wird eine Null-terminierte Zeichenkette verwendet

```
message3 byte   “hello world“, 0
```

DUP- Operator

- reserviert Speicher für mehrere Daten

```
db 10 dup (?)      ;10 Bytes, uninitialisert
db 10 dup (0)      ;10 Bytes, mit 0 initialisiert
```

8.3. Symbolische Konstanten

- besteht aus Bezeichner (Symbol) und Integer-Ausdruck oder Text
- werden vom Assembler nur während der Übersetzung verwendet und können zur Laufzeit nicht verändert werden

Beipiele:
```
monate = 12
mov eax, monate
```

Verwendung:
- wenn konstante Werte an verschiedenen Stellen im Programm verwendet werden

equ- Direktive

- ordnet einem symbolischen Namen einen Integer-Ausdruck oder einen Text zu
- eine Art Abkürzung oder anderer Name
- 3 Formen:

```
name equ  expression
name equ  symbol
name equ <text>
```

Beispiele:

```
pi    equ  <3.14159>
m1    equ  <“Press any key to continue“,0>
```

9. Struktur eines Assemblerprogramms (PM)

- 32-bit Programme kennen nur ein Speichermodell
  flat
- eigentlich gibt es keine verschiedenen Segmente mehr
- die Segmente werden aber in logische Abschnitte eingeteilt
- Kommunikation mit dem Betriebssystem erfolgt nicht über Interrupts (wie im RM) , sondern über die Win-API
- es gibt nur noch eine Aufrufkonvention: stdcall
  d.h.:
  - die Parameter werden von rechts nach links auf den Stack gelegt
  - die aufgerufene Funktion ist für die Stackbereinigung verantwortlich !

Mögliche Struktur

```
. 486
.model flat, stdcall
.data
.code
main: …
          …
end main
```

Bemerkung:

- Windows benutzt die Register ebx, edi, esi und ebp intern
- Register können verwendet werden, aber müssen zu Beginn der Prozedur eingekellert und vor Verlassen der Prozedur wieder ausgekellert werden

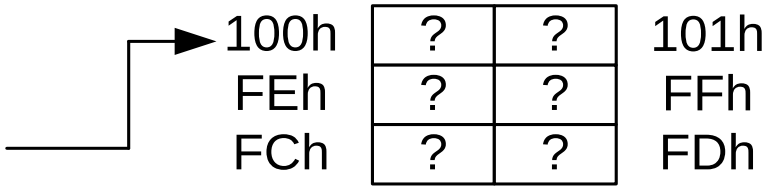
Zugriff auf das Stack-Segment

Prinzip: Last-In-First-Out
Sp zeigt auf die Adresse, an der da letzte auf dem Stack gespeicherte Wort abgelegt ist

push	Ablage Doppelwort im Stack
	- zuerst esp = esp - 4
	- dann Ablage Wort
pop	Holen Doppelwort aus dem Stack
	- zuerst Holen Wort
	- dann esp = esp + 4

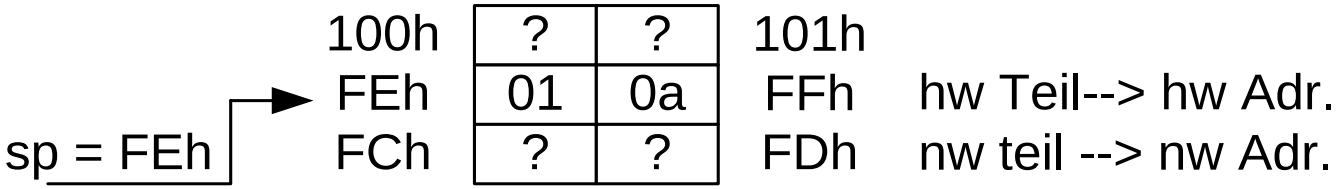
Beispiel:

ax = 0a01h
cx = 0250h
sp = 0100h



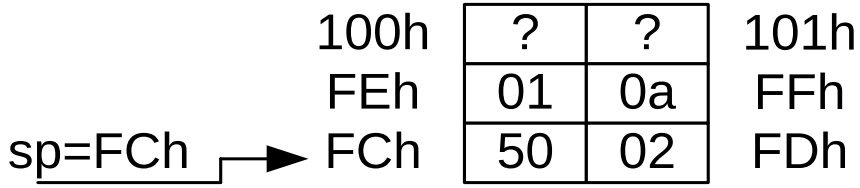
Stapel vor push ax

Ablegen im Stack: push ax



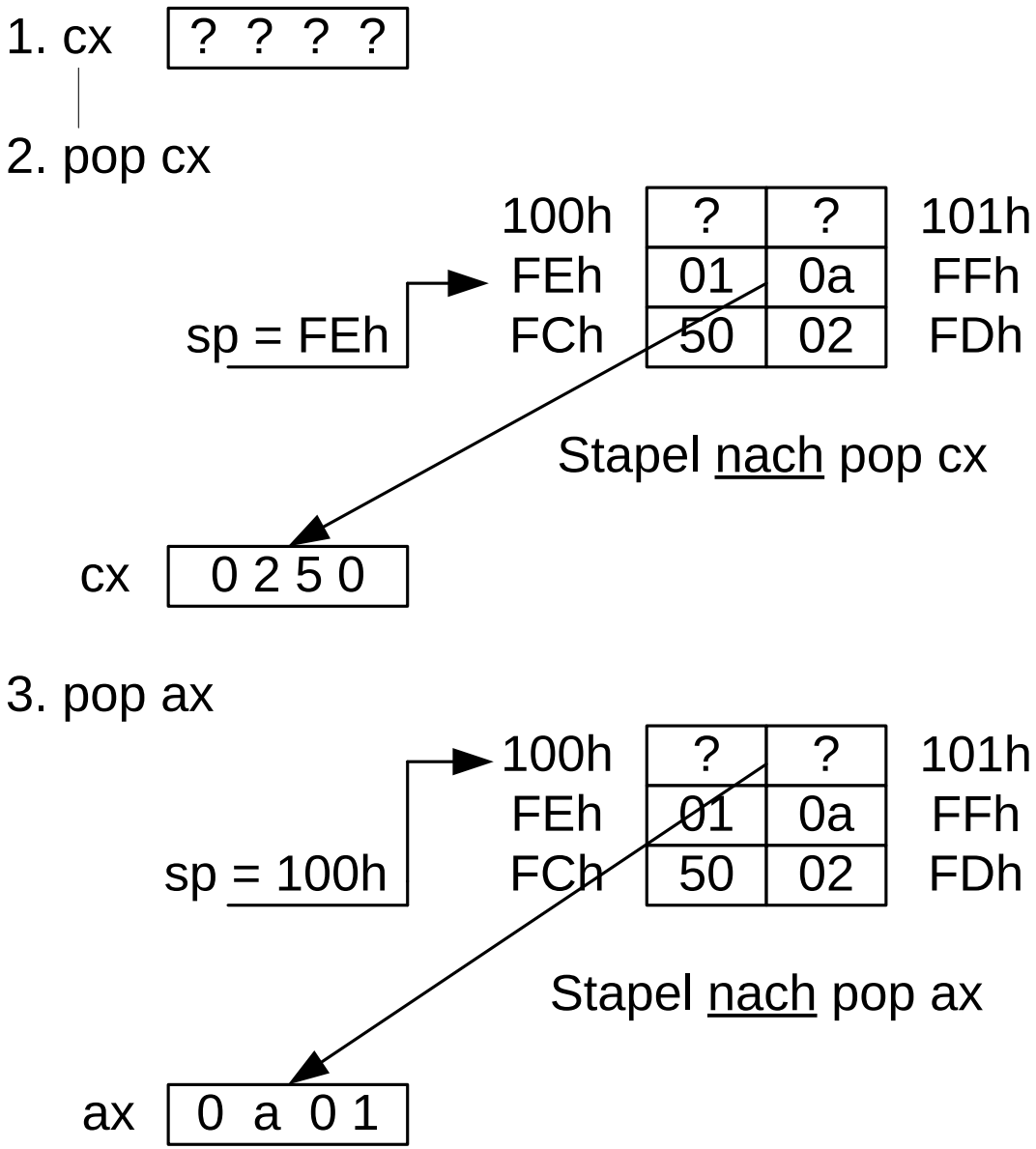
Stapel nach push ax

Ablegen im Stack: push cx



Stapel nach push cx

Holen aus Stack:



Reihenfolge:
- last – in – first – out (LIFO)

```
push ax
push cx
…
…
pop cx
pop ax
```

- alle Daten die im Stack auf diese Weise abgelegt werden, müssen auch wieder geholt werden
- nie den Inhalt des SP-Registers im Programm ändern (nur einmalig bei Initialisierung)

Zugriff auf das Datensegment

```
.data
var dw 0815h ; 16-Bit-Variable
…
.code
; die nächsten 2 Zeilen nur im RM!
mov ax, @data    ;ds muss mit der tatsächlichen
mov ds, ax       ;Basisadresse des Daten-
                 ;Segments geladen werden
;
…
mov ax, [var];    ax = 0815h
```

10. Befehlssatz

10.1. Übersicht über die Befehlsgruppen

- Datentransfer-Befehle
- Arithmetik-Befehle
- Logik-Befehle
- Stringmanipulations-Befehle
- Programmtransfer-Befehle
- Prozessorsteuerungs-Befehle

10.2. Datentransfer-Befehle

10.2.1. Transfer-Befehle für allgem. Anwendungen

mov dest, src ; dest := src

von →	nach
Register	Register
Register	Speicher
Speicher	Register
Datum	Register
Datum	Speicher
Seg.Reg.	Register
Register	Seg.Reg.

- Daten werden nicht verändert, keine Flags gesetzt

Beispiele: mov cx, 1000h
mov es, ax
mov byte ptr [bx], 0h

xchg Vertausche Quell- und Zieloperand
- Inhalt zweier Register
oder - Inhalt Register und Inhalt Speicherplatz
Beispiel: xchg di,si

push src Transportiere Wort in Stack
- zuerst SP:= SP -2
- dann Wort in Stack übertragen

pop dest Hole Wort aus dem Stack
- zuerst SP := Sp+2

10.2.2. Befehle zum Laden von Adressen

lea dest, src Lade effektive Adresse

- lädt Offset eines Speicheroperand. In ein 16-Bit-Register)

Beispiel: lea di, label

10.2.3. Befehle zur Flagübertragung

lahf Lade ah mit Flags

- kopiert S,Z,A,P,C in ah
- Flags bleiben unverändert

pushf Transportiere Flags in Stack

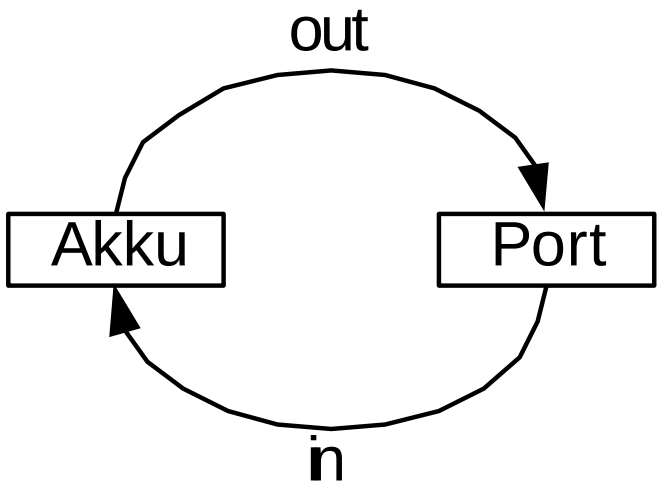
- wird bei ISR automatisch übertragen

popf Hole Flags aus Stack

- wird bei ISR automatisch übertragen

10.2.4. E/A-Befehle (nur RM!)

nur 2 Befehle zum Transfer zu/von Ports



Angabe der Portadresse:

- Portadresse 0-ffh → direkt im Befehl möglich
- Portadresse 0-ffffh → im Register dx

```
out dest, src        ;Byte- oder Worttransfer zum Port
→     out port_adr8, al        ;Ausgabe Byte
      out port_adr16, ax       ;Ausgabe Wort
      out dx, al               ;Ausgabe Byte
      out dx, ax               ;Ausgabe Wort

Beispiel:
mov al, …            ;Ausgabe Byte
out 0ch, al

mov ax, …            ;Ausgabe Wort
mov dx, …
out dx, ax
```

```
in dest, src          ;Einlesen Byte oder Wort vom Port
→     in al, port_adr8    ;Einlesen Byte
      in ax, port_adr8    ;Einlesen Wort
      in al, dx           ;Einlesen Byte
      in ax, dx           ;Einlesen Wort
```

10.3. Arithmetik-Befehle

10.3.1. Zahlenformate

8086
- vorzeichenlose 8-Bit-Binärzahl (0…255)
- vorzeichenlose 16-Bit-Binärzahl (0…65535)
- vorzeichenbehaftete 8-Bit-Binärzahl (-128…+127)
- vorzeichenbehaftete 16-Bit-Binärzahl (-32768…+32767)

80386 (ohne FPU)
zusätzlich:
- vorzeichenlose 32-Bit-Binärzahl ($0…2^{32} – 1$)
- vorzeichenbehaftete 32-Bit-Binärzahl ($-2^{31}…2^{31}-1$)

BCD-Zahlen 8086

- gepackt (0…99) $y_3y_2y_1y_0x_3x_2x_1x_0$
- als vorzeichenlose 8-Bit-Werte behandelt
- jedes Nibble ein BCD-Digit
- nur bei Addition und Subtraktion möglich
- ungepackt (0…9) $0000x_3x_2x_1x_0$
- als vorzeichenlose 8-Bit-Werte behandelt
- bei allen 4 Grundoperationen möglich

Flagbeeinflussung:

- bei Arithmetikoperationen werden Flags gesetzt

→ Befehlsliste

10.3.2. Addition

```
add dest, src       ;dest := dest + src
adc dest, src       ; dest := dest + src + c

Beispiele:   add al,bl
             add dx, 1000h

inc dest      ; dest := dest + 1

aaa           ;korrigiere Byte in ungepackter BCD-Zahl
              ;nach Addition
              ;vorher Byteaddition mit al als Ziel
daa           ;korrigiere Byte in gepackter BCD-Zahl
              ;nach Addition, vorher Addition der
                    ;beiden Dezimalzahlen mit al als Ziel
```

10.3.3. Subtraktion

```
sub dest, src       ;dest := dest – src
sbb dest, src       ;dest := dest – src – c
                    ;Subtraktion mit „borgen“
```

Beispiele:

```
sub al, bl
sub cx, 5
sub byte ptr[di], 100
```

```
dec dest            ;dest := dest – 1

neg dest            ;bildet Zweier-Komplement
```

cmp dest, src ;Vergleich Ziel- und Quelloperand

- Ausführung als Subtraktion
- y = dest – src ohne Ergebnis zu speichern
- Flags werden entsprechend der Subtraktion gesetzt

S	Z	C	
1	0	1	Ziel < Quelle
0	1	0	Ziel = Quelle
0	0	0	Ziel > Quelle

Beispiel: cmp ax, bx
cmp cl, 17h
cmp cx, 700h

aas ;korrigiere Byte in ungepackter BCD-Zahl
;nach Subtraktion
das ;… in gepackter BCD-Zahl

10.3.4. Multiplikation

mul src

1) Byte-Operand

src = Byte ax := src * al

2) Word-Operand

src = Word dx|ax := src * ax; dx signifikant,
wenn C = 1 und O = 1

3) Dword-Operand

src = Dword edx|eax := src * eax; edx signifikant,
wenn C = 1 und O = 1

Beispiele:

1) Byte-Operand

```
mov bl, 2
mov al, 3
mul bl          ; ax := bl * al = 6
```

2) Word-Operand

```
mov bx, 2
mov ax, 3
mul bx          ; dx|ax := bx * ax = 6
```

3) Dword-Operand

```
mov ebx, 2
mov eax, 3
mul ebx         ; edx|eax := ebx * eax = 6
```

10.3.5. Division

div src; Division vorzeichenloser Binärzahlen
  src = Divisor

Quotient = Dividend / Divisor

1) Byte-Operand

src = Byteoperand Divident: ah, al
  Quotient: al Rest: ah

div src ; al := ax : src

2) Word-Operand

src = Wortoperand Divident: dx (hw), ax (nw)
  Quotient: ax Rest: dx

div src ; ax := dx|ax : src

3) DWord-Operand

src = DWortoperand Divident: edx (hw), eax (nw)
Quotient: eax Rest: edx

```
div src        ; eax := edx|eax : src
```

Beispiel:

1) Word-Operand

```
mov dx, 0
mov ax, 6
mov bx, 3
div bx ; ax := dx|ax : bx = 2
```

2) Dword-Operand

```
mov edx, 0
mov eax, 6
mov ebx, 3
div ebx       ; eax := edx|eax : ebx = 2
```

Sonderfall:
Quotient bei Bytedivision > 255 oder
Quotient bei Wortdivision > 65535
Quotient bei Dworddivision > 2^{31} -1
--> Interrupt Typ 0 (Divisionsfehler)

10.4. Befehle zur Flagmanipulation

10.4.1. Flagbeeinflussung

- bei Bitmanipulationsbefehlen gelten folgende Regeln:
  1. C enthält Wert des letzten Bits, das aus dem Zieloperanden herausgeschoben oder –rotiert wurde
  2. O=1, wenn das höchstwertigste Bit des Zieloperanden seinen Wert ändert beim Verschieben um 1 Bitposition (sonst undefiniert)
  3. Z, S, P wie sonst
- alle Flags können mit bedingten Sprungbefehlen getestet werden

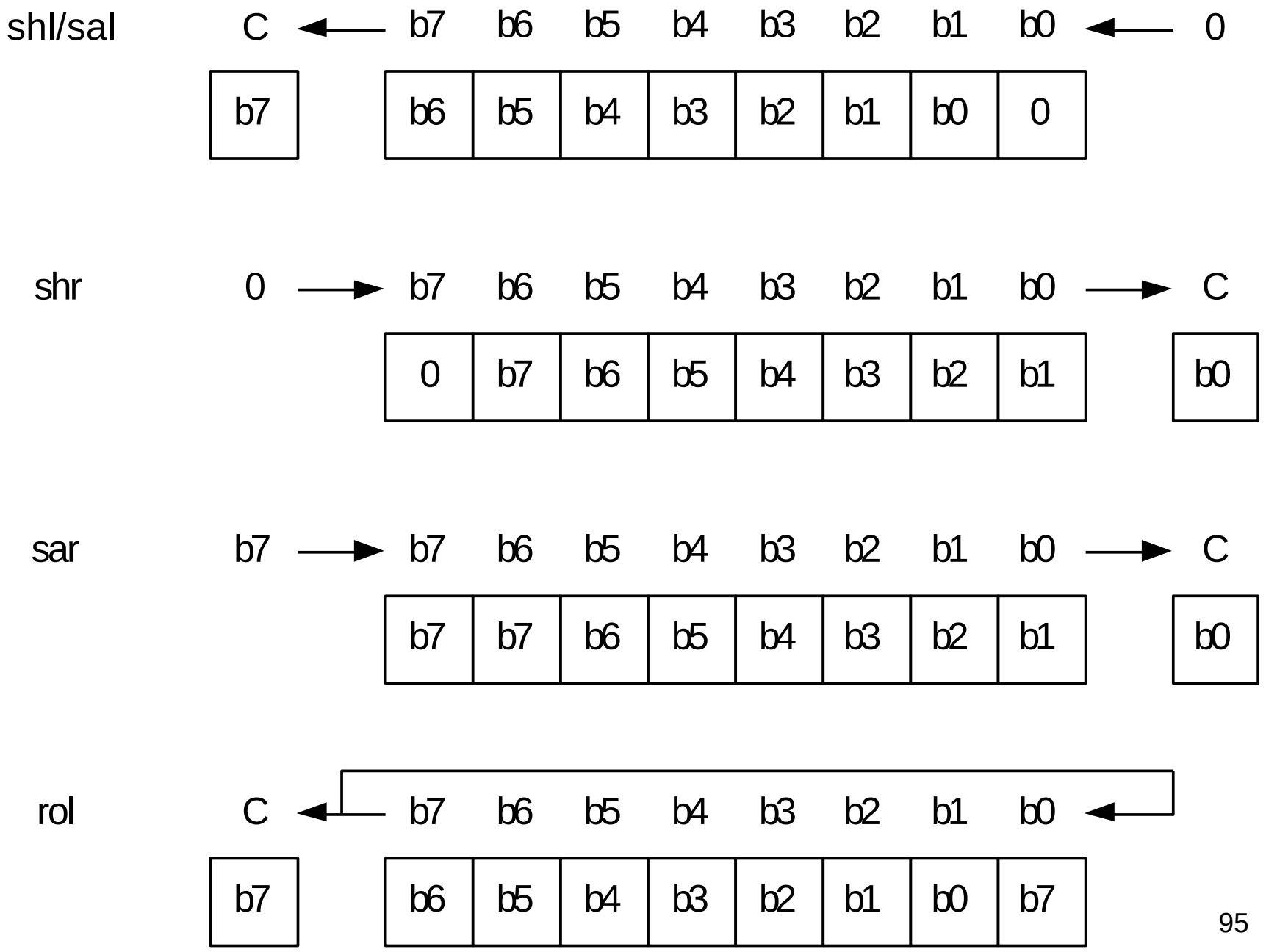
10.4.2. Befehle für logische Operationen

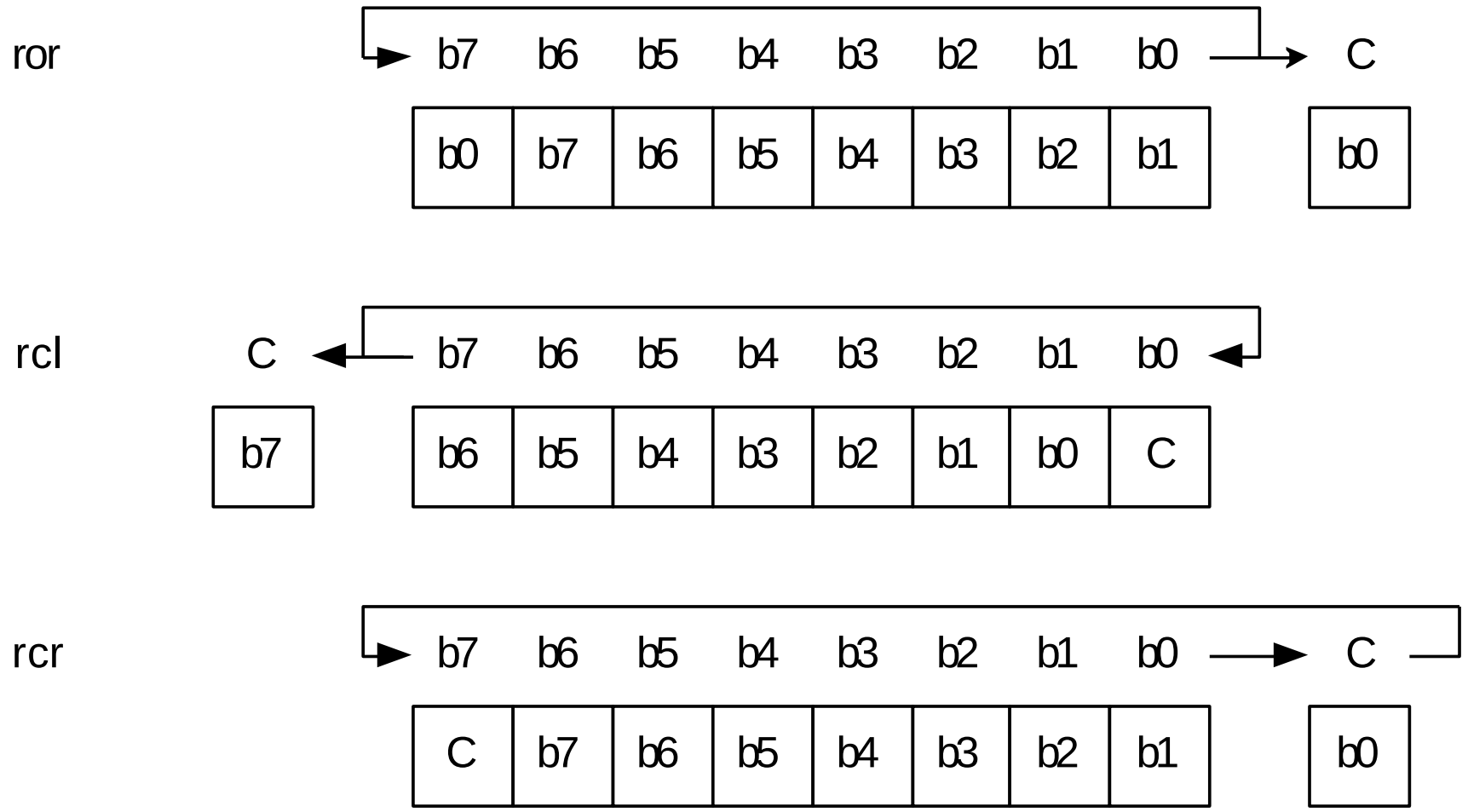
```
not dest      ;Einer-Komplement eines Register- oder
              ;Speicheroperanden

Beispiel:     mov ax, 5555h
              not ax                ;ax= aaaah

and dest, src       ;beide Operanden (Byte oder Wort)
                    ;werden bitweise UND-verknüpft

Beispiel:     mov bl, 10h           0001 0000 and
              mov bh, 14h           0001 0100
              and bl,bh             ----------
                                bl =  0001 0000= 10h
```

```
test dest, src       ;Vergleich durch logisches UND
                     ;keine Abspeicherung Operanden
                     ;Flagbeeinflussung

or dest, src         ;bitweise ODER-Verknüpfung
xor dest, src        ;bitweise Exklusiv-ODER-Verknüpf.
```

10.4.3. Bitschiebe- und Rotationsbefehle

- Verschieben von Bits in Speicher- oder Register-operanden
- Schiebebefehle: Bits, die herausgeschoben werden gehen verloren
- Rotationsbefehle: Bits werden im Kreise verschoben


shl/sal
C
b7 b6 b5 b4 b3 b2 b1 b0
0
b7
b6 b5 b4 b3 b2 b1 b0 0
shr
0
b7 b6 b5 b4 b3 b2 b1 b0
C
0 b7 b6 b5 b4 b3 b2 b1
b0
sar
b7
b7 b6 b5 b4 b3 b2 b1 b0
C
b7 b7 b6 b5 b4 b3 b2 b1
b0
rol
C
b7 b6 b5 b4 b3 b2 b1 b0
b7
b6 b5 b4 b3 b2 b1 b0 b7


ror
b7 b6 b5 b4 b3 b2 b1 b0 C
b0 b7 b6 b5 b4 b3 b2 b1 b0
rcl
C b7 b6 b5 b4 b3 b2 b1 b0
b7 b6 b5 b4 b3 b2 b1 b0 C
rcr
b7 b6 b5 b4 b3 b2 b1 b0 C
C b7 b6 b5 b4 b3 b2 b1 b0

10.5. Stringbefehle

Anwendung:
Manipulation von Zeichenketten (Strings) bis zu 64 KByte Länge

- Basisoperationen
  - Übertragungsbefehle
  - Vergleichsbefehle
  - Suchoperationen

  mit jeweils einem Stringelement
- Repeat-Präfix → Wiederholung der Operation, um Strings beliebiger Länge zu bearbeiten

  Vorteil: schneller als Software-Schleife

gemeinsame Kennzeichen aller Stringbefehle

- Adressierung prinzipiell über SI, DI
- Annahme: - Quellstring in DS (SI enthält Offset)
  Segment-Override-Präfix möglich
  - Zielstring in ES (DI enthält Offset)
  Segment-Override-Präfix möglich
- Anwendung:
  ganze Speicherbereiche mit einheitlicher Daten-
  struktur unter Einsatz weniger Befehle bearbeiten
  Funktionen: - Laden
  - Speichern
  - Vergleichen
  - Kopieren
  - Einlesen vom Port
  - Ausgabe an Port

Adressierung:

Befehl	Zieloperand	Quelloperand
movs	es : di	ds(es,ss,cs) : si
cmps	es : di	ds(es,ss,cs) : si
scas	es : di	ax / al
lods	ax / al	ds(es,ss,cs) : si
stos	es : di	ax / al

Beispiel: movs dest, src ;Übertrage Byte- oder
;Wortoperand
rep wiederhole, solange cx ≠ 0
(cx enthält zu Beginn Anzahl der Wiederhol.)

konkretes Beispiel:
Voraussetzung:
ds, es sind initialisiert
Var. dest_string und source_string sind vereinbart

```
mov si, offset source_string
mov di, offset dest_string
mov cx, length_source_string; Anzahl Byte
rep movs dest_string, source_string
```

10.6. Programmtransferbefehle

10.6.1. Wirkungsweise

- Programmierung von Sprüngen, Schleifen, Aufruf von Unterprogrammen
- Operanden müssen Adressinformation sein

2 Formen:

- direkt: Label (Marke) verweist auf Zielbefehl
- indirekt: Zieladresse ist in einem Zeiger enthalten
- 2 Typen von Operanden:
- near oder intrasegment
- short oder intrasegment-kurz

near
- Zieladresse im aktuellen Codesegment
- Operand: - 16 Bit oder 32 Bit
  - unmittelbar im Befehl oder
  - indirekt (Wort im Register oder Speicher)

short
- Zieladresse im Abstand -128…+127 Byte
- Operand: Byte, als vorzeichenbehaftete Zahl zu IP addiert
- nur direkte Form
- ausschließlich bei bedingten Sprungbefehlen und Schleifenbefehlen

10.6.2. Prozeduraufrufe

CALL rel16	Call UP (near), relativ
CALL rel32	Call UP (near), relativ
CALL r/m16	Call UP (near), absolut indirekt
CALL r/m32	Call UP (near), absolut indirekt
CALL ptr16:16	Call UP (far), absolut
CALL ptr16:32	Call UP (far), absolut
CALL m16:16	Call UP (far), absolut indirekt
CALL m16:32	Call UP (far), absolut indirekt

4 verschiedene Typen von „call“:
- near call
- Aufruf eines UP im aktuellen Code-Segment (aktuell durch das CS-Register adressiert)
- far call
- Aufruf eines UP in einem anderen Segment als dem aktuellen Code-Segment
- inter-privilege-level far call
- Aufruf eines far-UP in einem Segment mit einer anderen Privilegierungsstufe als das aktuelle Programm
- task switch
- Aufruf eines far-UP in einer anderen Task

10.6.2. Prozeduraufrufe

call procedure	;Aufruf UP (unbedingt)
- near call	EIP → Stack (RM, PM)
- far call	CS → Stack (RM)
	EIP → Stack
ret	;Rückkehr aus UP
- near UP	Stack → EIP (RM, PM)
- far UP	Stack → EIP (RM)
	Stack → CS

10.6.3. Sprungbefehle

unbedingte Sprünge:

- Ausführung immer
- Ziel: Typ far (nur im PM)/ near / short
- Operand: Marke
- Beispiel: jmp m1

bedingte Sprünge:

- Ausführung nur, wenn Bedingung erfüllt
- Ziel: Typ short
- Operand: Marke
- Beispiele:

```
jz m1        ; Springe, wenn Z = 1
jnz m2       ; Springe, wenn Z = 0
```

Testbedingungen für bedingte Sprünge

Bedingung	**Test**	**Befehl**
gleich	Z = 1	JE (JZ)
ungleich	Z = 0	JNE (JNZ)
größer	(S xor O) or Z = 0	JG / JNLE
kleiner	S xor O = 1	JL / JNGE
größer oder gleich	S xor O = 0	JGE / JNL
kleiner oder gleich	(S xor O) or Z = 1	JLE / JNG
über	C or Z = 0	JA / JNBE
unter	C = 1	JB / JNAE
über oder gleich	C = 0	JAE / JNB
unter oder gleich	C or Z = 1	JBE / JNA

Bedingung	**Test**	**Befehl**
Null	Z = 1	JZ (JE)
nicht Null	Z = 0	JNZ (JNE)
Übertrag	C = 1	JC
kein Übertrag	C = 0	JNC
Überlauf	O = 1	JO
kein Überlauf	O = 0	JNO
Vorzeichen negativ	S = 1	JS
Vorzeichen positiv	S = 0	JNS
Parität (gerade)	P = 1	JP
keine Parität (ungerade)	P = 0	JNP

10.6.4. Befehle zur Schleifensteuerung

- ecx/ cx als Schleifenzähler
- Schleifensteuerbefehl:
- dekrementiert ecx/ cx
- bedingter Sprung zu Zielbefehl (Marke),
  solange cx ≠0
- Operand: Marke im Abstand -128…+127 (short)

loop target ; Sprung zum Schleifenbeginn

Beispiel: m1: …

…

loop m1; Sprung, wenn cx ≠ 0

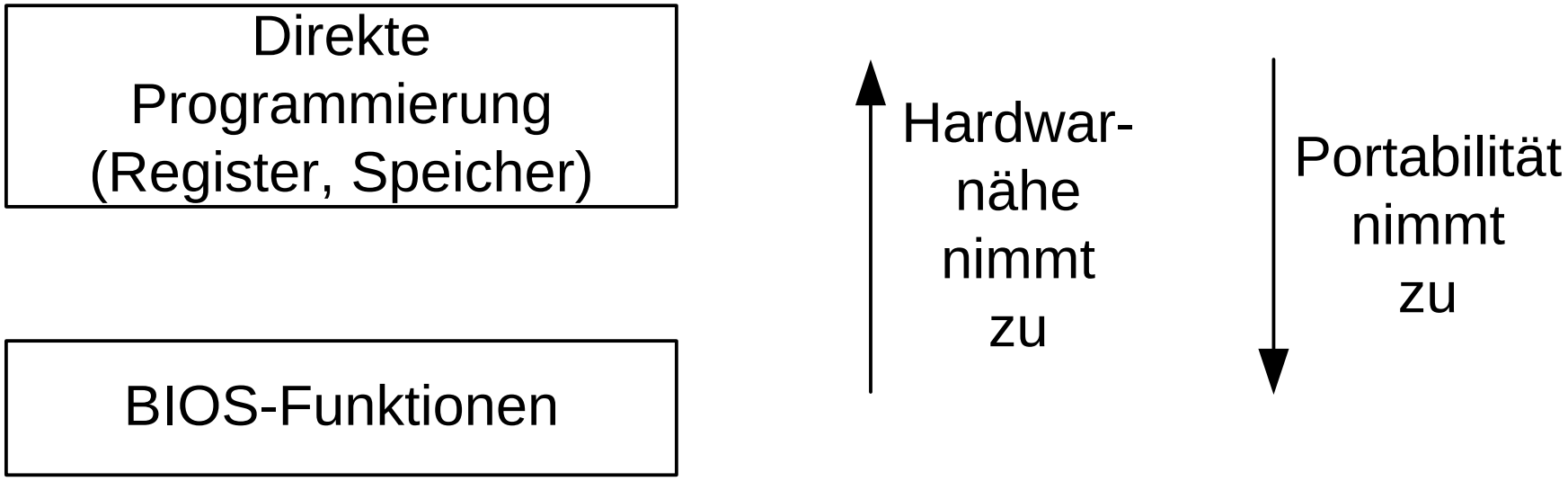
Varianten:

loope/loopz marke; Sprung, wenn cx ≠ 0 und z = 1

loopne/loopnz marke; Sprung, wenn cx ≠ = und z = 0

jcxz marke; Sprung, wenn cx = 0
; keine Dekrementierung von cx !
; ist cx zu Beginn gleich Null, wird die
; Schleife übersprungen

11. Aufruf von BIOS- Routinen (nur RM)



BIOS – Basic Input/Output-System
- umfangreiche Sammlung an Interrupt-Routinen
- für alle vorkommenden Zugriffe auf die Hardware
- nur im Real-Mode (DOS) bzw. beim Hochfahren des Rechners nutzbar, da nicht reentrant

allgemeines Prinzip:
Übergabe Parameter, Funktionsnummer → Register
Aufruf der (BIOS-) Interrupt-Routine
(Rückgabe von Parametern/Ergebnis, Cond.-Code)

Beispiel: Ausgabe auf dem Bildschirm

Interrupt: int 10h; Video E/A

Schritte:

a) Setze Video-Modus int 10h, Nr.0

ah = 0 (Nr. für Video-Modus setzen)

al = Nr. für Video-Modus

z.B. 80*25 alphanumerisch, s/w, Text

Beispiel:

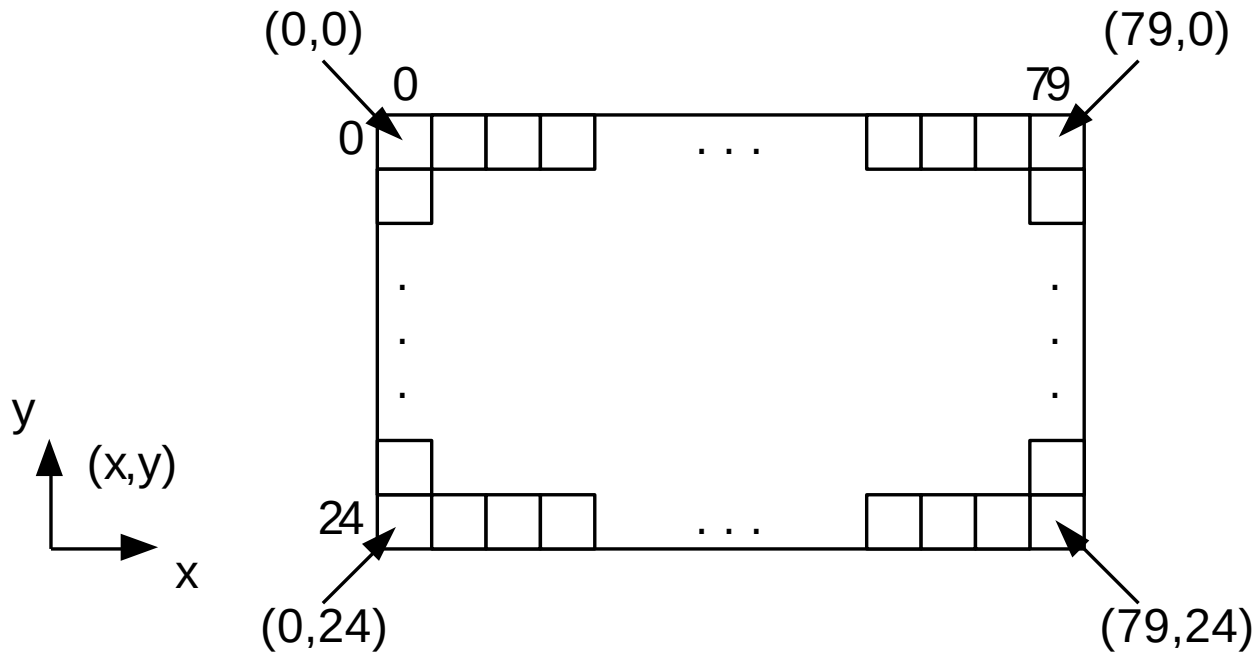
```
mov al, 2;   Nummer für obigen Modus
mov ah, 0;  Funktions-Nr. für Modus setzen
int 10h
```

b) Bildschirm löschen
(hier als Scrollen von Textzeilen) int 10h, Nr. 6

ah = 6 (Funktion-Nr.)
al = Anzahl Zeilem, die zu scrollen sind
= 0, löscht Bildschirm zwischen den Koordinaten
bh = Attribut
ch = y-Koord. linke, obere Ecke des Fensters
cl = x-Koord. linke, obere Ecke des Fensters
dh = y-Koord. rechte, untere Ecke des Fensters
dl = x-Koord. rechte, untere Ecke des Fensters

Beispiel: mov ah, 6

```
mov al, 0
mov cx, 0
mov dh, 24
mov dl, 79
mov bh, 07; schw. Char. auf schw. Grund
int 10h
```



c) Cursor setzen int 10h, Nr. 2

ah = 2
dh = Zeile
dl = Spalte
bh = aktive Seite

Beispiel:
```
mov ah, 2
mov dh, 12
mov dl, 35
mov bh, 0
int 10h
```

e) Ausgabe ASCII-Zeichen und Weiterrücken Cursor
um 1 Position int 10h, Nr.14

ah = 14
al = zu schreibendes Zeichen (ASCII)
bh = Bildschirmseite

```
Beispiel:     mov al, 1;Ausgabe einer „1“
              add al, 30h; 31h ASCII-Code für „1“
              mov ah, 14
              int 10h
```

12. Makro‘s, Includes, Unterprogramme, Bibliotheken

12.1. Makro‘s

Makro

- Programmabschnitt, der nach seiner Definition im Programm im folgenden Programmtext bei jedem Auftreten eingefügt wird (und dort dann auch assembliert wird) → „inline expansion“
- gibt keinen Wert zurück
- werden definiert:
    - am Anfang des Quellcodes (vor Einsprungspunkt) oder
    - über include- Direktive

Vergleich Makro mit UP
Nachteil: Code wird länger
Vorteil: Code (als .exe) wird schneller

Definition

```
name macro      ;Beginn Makro
        …       ;Programmtext
        endm    ;Ende Makro
```

- Übergabe von Parameter möglich

```
mac1 macro param_1, param_2[, param_3,..]
        …
        endm
```

- Aufruf Makro

;nach Makrodefinition!

mac1 2,5,10 ;2,5,10 sind 3 Parameter

- Vorzeitiges Verlassen des Makros (vor Ende)

exitm

Kommentare in Makros

- normale Kommentare in Makro- Definition
  erscheinen bei jeder Expansion
- soll Kommentar nicht bei der Expansion erscheinen
  „;;“

ECHO- Direktive

- zeigt Nachrichten auf der Konsole an, wenn das
  Programm assembliert wird

Beispiel:

```
macro1 MACRO
        ECHO Makro1 wird expandiert
        …
        …
        ENDM
```

12.2. Include-Dateien

- Anweisung: include
- Möglichkeit, bestimmte Programmteile in Quelltext einzufügen
- über include eingefügte Quelltext-Datei darf nicht in Projektmappe stehen (z.B. nicht unter Quelldateien!)

…

include teil1.asm; Datei teil1 wird eingefügt
include c:\work\teil1.asm

…

- include-Dateien können wiederum geschachtel werden (beliebige Tiefe)

12.3. Unterprogramme

- ähnlich aufgebaut wie Makro‘s
- aber grundsätzlicher Unterschied beim Aufruf

Makro-Aufruf:
Anweisung an Assembler, alle Befehle des Makro‘s zur Aufrufzeit an die Aufrufstelle zukopieren

UP-Aufruf:
zur Ausführungszeit des Programms Sprung an die Stelle, an der das UP steht
vorher Abspeicherung der Adresse, die dem call-Befehl folgt

Definition UP

```
name proc    [near|far]
        …
        ret
     [name] endp
```

Aufruf UP

call adress_ausdruck; i.a. Name

Beispiel:

call einzahl

- UP-Definition im Programm-Teil, vor Hauptprogr.
- es muss verhindert werden, dass man ohne Aufruf in das UP gelangt

```
.code
up1 proc
        ...
        ret
up1 endp


main:
;                       hier steht
 call up1       ;       eigener
;                       Code
exit
end main
```

call und ret

call

- schafft die Rückkehradresse auf den Stack
- lädt die (Anfangs-)Adresse in EIP

ret

- holt die Rückkehradresse vom Stack und lädt sie in EIP

Labels in Prozeduren

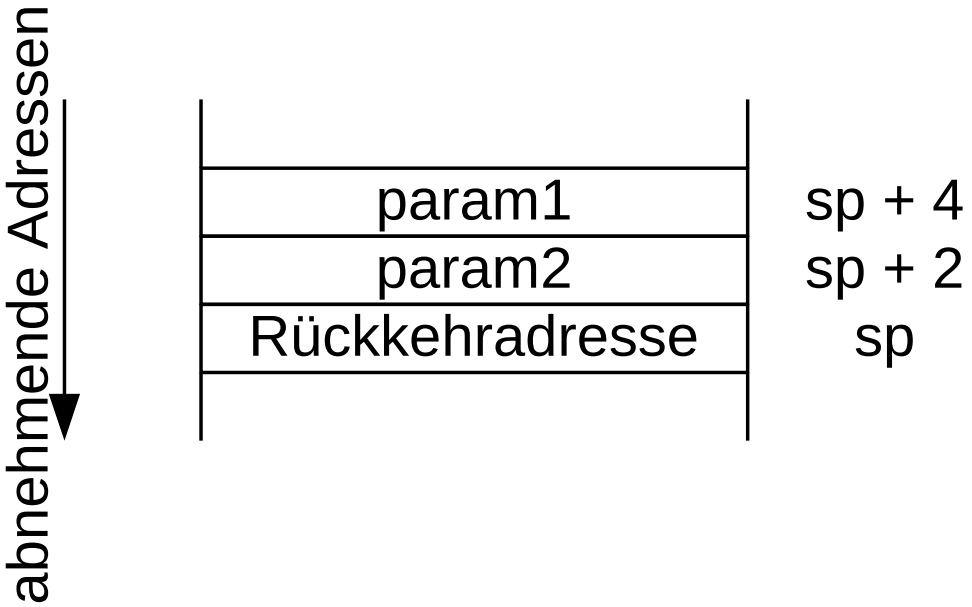
- gelten nur innerhalb der Prozedur (z.B. bei Sprüngen oder Schleifen)

Parameter-Übergabe

a) über Register
Vorteil:
- sehr einfach, häufig angewandt
Nachteil:
- bei jedem UP neue Festlegung
- Festlegung muss genau eingehalten werden
- nur bei wenigen Parametern möglich

b) auf dem Stack
- Parameter-Übergabe vor dem UP-Aufruf

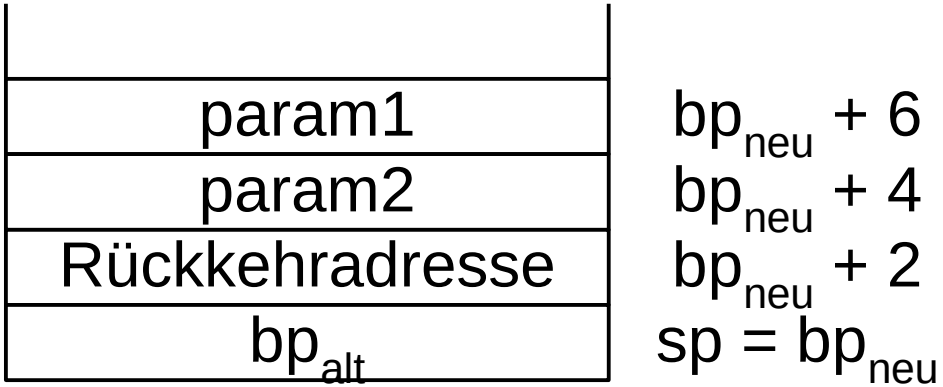
Beispiel:
mov ax, param1
push ax
mov ax, param2
push ax
call up1



- im UP am Anfang

```
push bp      ;Inhalt bp → Stack
mov bp, sp ;neuer Inhalt von bp zeigt auf die
             ;Stack-Adresse, an der der alte
             ;Wert von bp steht
```

Keller jetzt



- werden weitere Daten auf den Stack gebracgt,
  ändert sich nur sp;
  bp muss innerhalb des UP unverändert bleiben
- über bp kann im UP auf die aktuellen Parameter
  indirekt zugegriffen werden

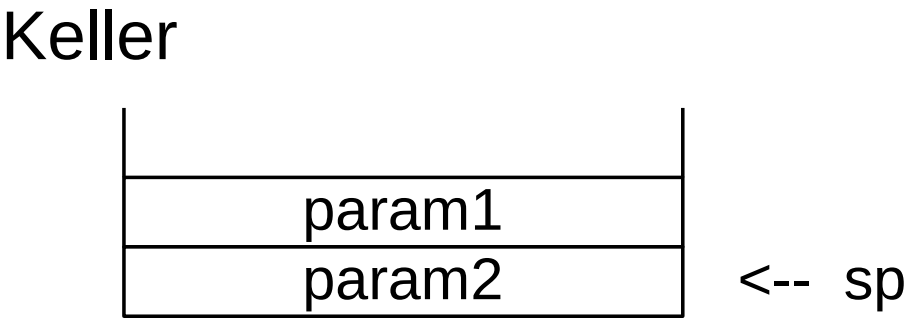
[bp+6] auf Parameter1
[bp+4] auf Parameter2

- vor Verlassen UP

…

```
mov sp, bp
pop bp
ret
endp        ; Ende des UP
```

- nach Rückkehr aus UP (im HP nach call-Befehl)



- durch 2 pop-Befehle könnte der Stack gelöscht werden → umständlich
- besser:
  im UP: ret 4
  optional löscht 4 Byte vom Stack (= 2 Worte!)

Parameter-Rückgabe

Problem:
Aufrufer muss nach abarbeiten des UP noch zugreifen können
→ - Speicherplatz für Ergebnis(se) vor den Parameter des UP reservieren
- diesen Speicherplatz beim ret-Befehl nicht mit freigeben
- Aufrufer kann mit pop Befehl den Ergebnis-Wert vom Stack holen

13. Verwendung des Inline-Assemblers in Visual Studio 2017

13.1. Vorteile des Inline-Assemblers

- Optimieren geschwindigkeits-kritischer Code- Abschnitte
- Reduzierung der Code-Größe
- Zugriff auf Hardware (für Gerätetreiber)
- Nutzung von C / C++- Befehlen ohne zusätzliches Assemblieren und Linken
- zusätzlicher Assembler wie MASM32 wird nicht benötigt

aber:

- unterstützt nicht alle Makro- und Daten-Direktiven des MASM
- keine Unterstützung für Itanium und x64- CPU's

13.2. Das Schlüsselwort _asm

- kann dort verwendet werden, wo auch eine C- oder C++ - Anweisung stehen kann
- dem Schlüsselwort _asm muss folgen:
  - eine Assembleranweisung oder
  - ein Block mit Assembler-Anweisungen

Beispiele:

1. `_asm mov dx, ax`
2.
```
_asm
        {
        mov ax, bx
        …
        }
```

- Klammern um _asm-Block haben keinen Einfluss auf Gültigkeitsbereich von Variablen!

13.3. Befehlssatz, Datendirektiven, Variablen, Operatoren

Befehlssatz
- es wird der komplette Befehlssatz des Pentium 4 bzw. AMD Athlon unterstützt
- mit der Pseudoanweisung _emit können weitere Befehle implementiert werden, die der Zielprozessor unterstützt

MASM-Ausdrücke
- jeder beliebige MASM-Ausdruck kann verwendet werden (jede Kombination von Operanden und Operatoren, die zu einem Wert oder einer Adresse ausgewertet werden)

Datendirektiven

- ein _asm-Block kann auf C- oder C++-Datentypen und C- oder C++Objekte verweisen
- ein _asm-Block darf keine Datenobjekte mit MASM-Direktiven oder MASM-Operatoren definieren; d.h. es sind z.B. nicht erlaubt
    - Direktiven db, dd, dw, …
    - Operatoren dup, this
- nicht erlaubt sind die Direktiven
    - struc
    - record
    - width
    - mask

Makros
- dürfen nicht benutzt werden, speziell die Direktiven
    - macro, rept, irc, irp, endm
- auch Makro- Operatoren sind nicht zulässig, wie
    - <>, !, &, .type
- in einem _asm- Block dürfen C- Preprozessor-Direktiven verwendet werden

Kommentare
- im asm-Block kann „;“ verwendet werden
- innerhalb von Makros sollte auf Assembler-Kommentare verzichtet werden
- Kommentare in C-Art sind möglich

Debugging

- Inline- Assembler- Code kann mit der Option /Zi debuggt werden
- im Debugger können Haltepunkte sowohl auf C/C++- Zeilen als auch Inline- Assembler- Zeilen gesetzt werden

Sprungmarken

- Gültigkeitsbereich einer Sprungmarke: gesamte Funktion, in der sie definiert wurde
- mit Assembler-Befehlen als auch goto-Anweisungen kann zu Sprungmarken innerhalb oder außerhalb des _asm-Blocks gesprungen werden
- es dürfen keine Namen von C-Bibliotheks-Funktionen als Sprungmarke verwendet werden

Beispiele:

```
void main()
{       int x=0;
_asm
{
                mov ax, 4
                jmp m1
                nop
                mov bx,5
}
m1:   x= x + 1;
}
```

```
void main()
{       int x=0;
x = x + 1;
goto m1;
_asm
{
                mov ax, 4
                nop
}
m1:;
_asm
{
        mov bx,5
}
x= x + 1;
}
```

Verwenden von C- /C++ im _asm-Block

- C- und C++- Variablen können mit ihrem Namen angesprochen werden
- weitere nutzbare C-Sprachelemente:
    - Symbole (Sprungmarken, Variablen, Funktionsnamen)
    - Konstanten (z.B. symbolische Konstanten)
    - Makros und Präprozessor-Direktiven
    - Kommentare (/*…*/; //)
    - Typnamen (dort, wo ein MASM-Typ erlaubt wäre)
    - typedef-Namen
    - Ganzzahl-Konstanten können auch in C-Notation (z.B. 0x200) angegeben werden

Aufruf von C-Funktionen im _asm-Block

- C-Funktionen sind aufrufbar, auch C-Bibliotheks-Funktionen;
  z.B. Verwendung der Funktion printf:
  - die Übergabe der Funktionsargumente erfolgt auf dem Stack
  - im Beispiel müssen Zeiger auf Zeichenfolgen vor
  - dem Funktionsaufruf mit push auf den Stack gelegt werden

Beispiel: Emulation der Funktion
printf(format, hello, world);

```
#include<stdio.h>
char format[] = “%s %s\n“;
char hello[] = “Hello“;
char world[] = “world“;
void main(void)
{
_asm
{
        mov eax, offset world
        push eax
        mov eax, offset hello
        push eax
        mov eax, offset format
        push eax
        call printf                     ;Aufruf der C-Funktion
        //Stack säubern
        pop ebx
        pop ebx
        pop ebx
}
}
```

Aufruf von C++-Funktionen im _asm-Block

- nur globale Funktionen aufrufbar, die nicht überladen sind
- es sind auch Funktionen aufrufbar, die mit extern “C“ für C-Bindung deklariert sind

Definieren von _asm-Blöcken als C-Makros

- empfehlenswerte Methode, um Assemblercode in C-/C++-Quellcode zu integrieren
- aber Makro wird zu einer einzigen logischen Zeile erweitert
- Makro kann Parameter haben, aber keinen Ergebniswert
- Regeln zum Schreiben eines solchen Makros
    - _asm{ …} verwenden
    - als Kommentar nur /*…*/ verwenden
    - jeder Zeile _asm voranstellen

Beispiel /6/:

```
#define PORTIO _asm
/* Port-Ausgabe */
{
_asm mov al, 2
_asm mov dx, 0x278h
_asm out dx, al
}
```

Erweiterung zu einer einzigen Zeile

```
_asm/* Port-Ausgabe */ { _asm mov al, 2 _asm mov dx, 0x278h _asm dx, al }
```

Verwenden und Sichern von Registern

- es ist unsicher,
    - ob ein Register zu Beginn des _asm-Block einen bestimmten Wert hat
    - ein Registerwert über verschiedene _asm-Blöcke hinweg seinen Wert behält
- eax, ebx, ecx, edx, esi, edi müssen im _asm-Block nicht gerettet werden alle anderen Register müssen im Gültigkeitsbereich des _asm-Blocks gerettet werden
- auch das Richtungsflag-Bit D muss bei Bedarf gerettet werden (darf seinen Wert nicht ändern)

Optimieren von Inline-Assembler

- Compiler versucht nicht, den _asm-Block zu optimi-
  mieren
- wird ein _asm-Block verwendet, versucht der
  Compiler nicht die Variablen in Register zu
  speichern, wenn die Gefahr besteht, dass sich die
  Registerwerte im _asm-Block ändern

Literatur:

1.
Intel 64 and IA-32 Architectures Software Developer‘s Manual
Volume 3A: System Programming Guide, Part 1

2.
Intel 64 and IA-32 Architectures Software Developer‘s Manual
Volume 3B: System Programming Guide, Part 2

3.
Diederich, Ernst-Wolfgang
Oldenbourg, 2000

4. Roming, Marcus; Rohde, Joachim
Assembler
mitp, 2003

5.
Kip R. Irvine
Assembly Language for x86 Processors
6. edition, Prentice Hall