

# Adressierung

Mit der Einführung des i80386 standen 32-Bit-Register zur Verfügung. Mit einem 32-Bit-Zeiger kann man  $2^{32} = 4\text{GB}$  Speicher adressieren. Jede Adresse wird durch einen einzigen 32-Bit-Wert beschrieben, den 32-Bit-**Offset**.

## Adressierungsarten

### Unmittelbare Adressierung

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

Beispiele:

```
MOV BX, 9           ; 9 wird unmittelbar ins Register bx geschrieben
BYTE_VALUE DB 150   ; Ein Byte-Wert (BYTE_VALUE) wird zum Wert 150
                    definiert
ADD BYTE_VALUE, 65   ; Es wird unmittelbar der Wert 65 addiert
MOV AX, 45H          ; 45H wird unmittelbar ins Register AX geschrieben
```

### Registeradressierung

Der Registeradressierung sind Quelle und Ziel interne Register des Prozessors.

Beispiele:

```
MOV EAX, EBX
```

### Direkte Speicheradressierung

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

Beispiele:

```
MOV AX, Zaehler1 ;Direkte Adressierung ohne eckige Klammern
MOV [bigcount], ECX ;Direkte Adressierung mit eckigen Klammern
MOV ECX, [Feld+2] ;Direkte Adr. von Feld + 2 Byte
```

## Indirekte Speicheradressierung

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

Nehmen wir z.B. folgende Aufgabenstellung: Für eine Zeichenkette soll die Häufigkeit der darin vorkommenden Zeichen bestimmt werden. Man braucht ein weiteres Feld um die Häufigkeit jedes Zeichens abzuspeichern. Bei der Bestimmung der Häufigkeit muss für jedes erkannte Zeichen der dazu gehörende Häufigkeitszähler um eins erhöht werden. Auf welchen Speicherplatz zugegriffen wird, ergibt sich also erst zur Laufzeit und hängt von den Daten ab. Eine direkte Adressierung, wie z.B.

```
inc [Häufigkeit+5]
```

reicht nicht aus. Ebenso liegt der Fall bei der Programmierung von Sortieralgorithmen und vielen anderen Problemstellungen der Informatik.

Bei dem Problem der Häufigkeitsbestimmung wäre nach den Deklarationen

```
Zeichenkette DB 'ABCDEFGH'  
Haeufigkeit DB 26 DUP (0)
```

im Codesegment eine direkte Adressierung wie z.B.

```
INC [Haeufigkeit+3]
```

nicht zweckmäßig, sie würde immer das Feldelement 3 (das vierte) ansprechen. Man müßte statt der 3 etwas Variables einsetzen können, je nachdem für welchen Buchstaben man den Zähler inkrementieren möchte.

Genau dies erlaubt die Register-indirekte Adressierung oder kurz indirekte Adressierung:

```
MOV BX, 3 ;Vorbereitung  
INC [Haeufigkeit+BX] ;indirekte Adressierung
```

wird nun auch das Feldelement 3 angesprochen, hier kann man aber zur Laufzeit berechnen, welcher Speicherplatz angesprochen werden soll, indem man BX mit diesem Wert vorbelegt, kann dann auch der 4. oder 10. Platz angesprochen werden.

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

### [Basisregister + Indexregister + Konstanten]

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

**(A1)**

Was macht der folgende Assembler Code? Beschreibe, was deiner Meinung nach geschieht und auf der Konsole ausgegeben wird, teste anschließend.

```
section .data
tabelle TIMES 10 DW 97

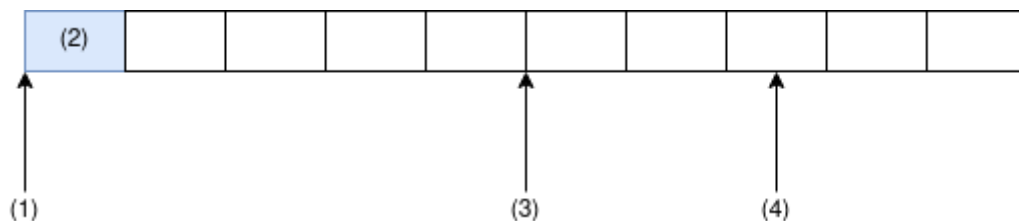
section .text
global _start
_start:

    mov     edx,20
    mov     ecx,tabelle
    mov     ebx,1
    mov     eax,4
    int     :w 0x80

    mov     eax,1
    int     0x80
```

**(A2)**

Das folgende Bild zeigt ein Speicherschema zum obigen Code.



Beantworte die folgenden Fragen:

- Wie groß ist der blaue Speicherbereich (2)?
- Welche Adresse hat die mit (1) bezeichnete "Speicherzelle"?
- Welche Adresse hat die mit (3) bezeichnete "Speicherzelle"?

Mache dir klar, dass die Adressierung durch Offsets erfolgt, weil vor dem Start des Programms nicht feststeht, an welchen absoluten Speicheradressen Daten und Programm bei der Ausführung geladen

werden! Man kann sich also vorstellen, dass bei der Deklaration einer Variablen wie `tabelle` eine neue "Nulladresse" gespeichert wird, von der aus man sich dann z.B. Byteweise durch den Speicher bewegen kann.

---



### (A3)

Was macht der folgende Code? Verfahre wie oben. Erläutere, was an den mit EINS und ZWEI markierten Stellen passiert.

```
section .data
tabelle TIMES 10 DW 97

section .text
    global _start
_start:

    mov EBX, tabelle    ; EINS
    mov [EBX], word 98 ; ZWEI

    mov edx, 20
    mov ecx, tabelle
    mov ebx, 1
    mov eax, 4
    int :w 0x80

    mov eax, 1
    int 0x80
```



### (A4)

Was macht der folgende Code? Verfahre wie oben. Ergänze nach den Semikola erklärende Kommentare

```
section .data
tabelle TIMES 10 DW 97

section .text
    global _start
_start:
```

```
mov EBX, tabelle ;  
mov [EBX+4], word 98 ;  
ADD EBX, 4 ;  
MOV [EBX], word 99 ;
```

```
mov edx, 20  
mov ecx, tabelle  
mov ebx, 1  
mov eax, 4  
int :w 0x80
```

```
mov eax, 1  
int 0x80
```

From:  
<https://www.info-bw.de/> -

Permanent link:  
<https://www.info-bw.de/faecher:informatik:oberstufe:techinf:assembler:adressierung:start?rev=1632162695>

Last update: **20.09.2021 18:31**

