**Curs 5 şi 6**

Contents

[1. Elementele de bază ale limbajului de asamblare 3](#_Toc18488386)

[1. Die Grundelemente der Assemblersprache 3](#_Toc18488387)

[1.1 Limbaj simbolic. Simboluri. Mnemonice şi etichete 3](#_Toc18488388)

[1.1 Symbolsprache. Symbole. Mnemonik und Beschriftungen 3](#_Toc18488389)

[1.2 Formatul unei linii sursă 4](#_Toc18488390)

[1.2 Das Format einer Quellzeile 4](#_Toc18488391)

[2. Expresii 6](#_Toc18488392)

[2. Ausdrücke 6](#_Toc18488393)

[2.1 Moduri de adresare 7](#_Toc18488394)

[2.1 Adressierungsarten 7](#_Toc18488395)

[2.1.1 Utilizarea operanzilor imediaţi 7](#_Toc18488396)

[2.1.1 Verwendung von Sofortoperanden 7](#_Toc18488397)

[2.1.2 Utilizarea operanzilor registru 10](#_Toc18488398)

[2.1.2 Verwendung von Registrierungsoperanden 10](#_Toc18488399)

[2.1.3 Utilizarea operanzilor din memorie 10](#_Toc18488400)

[2.1.3 Verwenden der Operanden aus dem Speicher 10](#_Toc18488401)

[2.2 Utilizarea operatorilor 14](#_Toc18488402)

[2.2 Verwendung von Operatoren 14](#_Toc18488403)

[2.2.1 Operatori de deplasare de biţi 15](#_Toc18488404)

[2.2.1 Bitweise Verschiebungoperatoren 15](#_Toc18488405)

[2.2.2 Operatori logici pe biţi 15](#_Toc18488406)

[2.2.2 Bitweise Logischeoperatoren 15](#_Toc18488407)

[2.2.3 Operatorul de specificare a segmentului 16](#_Toc18488408)

[2.2.3 Der Segmentspezifikationsoperator 16](#_Toc18488409)

[2.2.4 Operatori de tip 16](#_Toc18488410)

[2.2.4 Typoperatoren 16](#_Toc18488411)

[3. Directive 18](#_Toc18488412)

[3. Anweisungen 18](#_Toc18488413)

[3.1 Directiva SEGMENT 18](#_Toc18488414)

[3.1 Die SEGMENT-Anweisung 18](#_Toc18488415)

[3.2 Directiva ASSUME 20](#_Toc18488416)

[3.2 Die ASSUME-Anweisung 20](#_Toc18488417)

[3.3 Directive pentru definirea datelor 22](#_Toc18488418)

[3.3 Anweisungen zur Datendefinition 22](#_Toc18488419)

[3.4 Directiva EQU 25](#_Toc18488420)

[3.4 Die EQU-Anweisung 25](#_Toc18488421)

[3.5 Directivele LABEL şi PROC 26](#_Toc18488422)

[3.5 LABEL- und PROC- Anweisungen 26](#_Toc18488423)

[3.6 Blocuri repetitive 28](#_Toc18488424)

[3.6 Wiederholungsblöcke 28](#_Toc18488425)

[3.7 Directiva INCLUDE 30](#_Toc18488426)

[3.7 Die INCLUDE-Anweisung 30](#_Toc18488427)

[3.8 Macrouri 31](#_Toc18488428)

[3.8 Makros 31](#_Toc18488429)

|  |  |  |
| --- | --- | --- |
| 1. Elementele de bază ale limbajului de asamblare |  | 1. Die Grundelemente der Assemblersprache |
| ***Limbajul de asamblare*** al unui calculator este un limbaj de programare în care setul de bază al instrucţiunilor coincide cu operaţiile maşinii şi ale cărui structuri de date coincid cu structurile primare de date ale maşinii.  ***Limbajul maşină*** al unui sistem de calcul (SC) este format din totalitatea instrucţiunilor maşină puse la dispoziţie de procesorul SC. Acestea se reprezintă sub forma unor şiruri de biţi cu semnificaţie prestabilită. |  | ***Die Assemblersprache*** eines Computers ist eine Programmiersprache, in der der grundlegende Anweisungsatz mit den Operationen der Maschine übereinstimmt und deren Datenstrukturen mit den primären Datenstrukturen der Maschine übereinstimmen. ***Die Maschinensprache*** eines Computersystems (CS) besteht aus allen vom CS-Prozessor bereitgestellten MaschinenAnweisungen. Diese werden in Form von Bitfolgen mit vorgegebener Signifikanz dargestellt. |
| Limbaj simbolic. Simboluri. Mnemonice şi etichete |  | 1.1 Symbolsprache. Symbole. Mnemonik und Beschriftungen |
| Elementele cu care lucrează un asamblor sunt:   * etichete – nume scrise de utilizator, cu ajutorul cărora se pot referi date sau zone de memorie. * instrucţiuni – scrise sub forma unor mnemonice care sugerează acţiunea. Asamblorul generează octeţii care codifică instrucţiunea respectivă. * directive – sunt indicaţii date asamblorului în scopul generării corecte a octeţilor. Exemplu: relaţii între modulele obiect, definirea unor segmente, indicaţii de asamblare condiţionată, directive de generare a datelor. * contor de locaţii (*program counter*) – număr întreg gestionat de asamblor. În fiecare moment, valoarea contorului coincide cu numărul de octeţi generaţi corespunzător instrucţiunilor şi directivelor deja întâlnite în cadrul segmentului respectiv (deplasamentul curent în cadrul segmentului). Programatorul poate utiliza această valoare (accesare doar în citire!) prin simbolul '$'.   Asamblorul suportă două simboluri speciale („*tokens*”) în expresii, cu ajutorul cărora se pot efectua calcule care implică poziţia curentă la care a ajuns execuţia programului scris în limbaj de asamblare: este vorba despre „$” şi „$$”. *Token*-ul „$” reprezintă poziţia curentă a liniei care conţine expresia cu „$”. Deci putem codifica o buclă infinită dacă scriem „JMP $”.  *Token*-ul „$$” reprezintă începutul secţiunii curente. Aşadar, putem determina cât de departe am ajuns în interiorul secţiunii folosind expresia „($-$$)”. |  | Die Elemente, mit denen eine Assembler arbeitet, sind:   * Label – vom Benutzer geschriebene Namen, mit denen auf Daten oder Speicherbereiche verwiesen werden kann. * Anweisungen – in Form von Mnemonik geschrieben, die die Aktion vorschlagen. Der Assembler generiert die Bytes, die den jeweiligen Anweisung codieren. * Direktiven – sind Angaben zur Assembler, um die korrekten Bytes zu generieren. Beispiel: Beziehungen zwischen Objektmodulen, Definition von Segmenten, bedingte Assemblierungsanweisungen, Anweisungen zur Datengenerierung. * Positionszähler, Programmzähler oder Befehlszähler (*program counter*) – ganze Zahl, die von der Assembler verwaltet wird. Zu jedem Zeitpunkt stimmt der Wert des Programmzählers mit der Anzahl der Bytes überein, die gemäß den Anweisungen und Anweisungen generiert wurden, die bereits in dem jeweiligen Segment angetroffen wurden (die aktuelle Offset innerhalb des Segments). Der Programmierer kann diesen Wert (Nur-Lese-Zugriff!) mit dem Symbol '$' verwenden.   Der Assembler unterstützt zwei spezielle Symbole („*Tokens*“) in den Ausdrücken, mit denen Berechnungen durchgeführt werden können, die die aktuelle Position implizieren, an der die Ausführung des geschriebenen Programms in Assemblersprache angekommen ist: „$“ und „$$“. Das *Token* „$“ repräsentiert die aktuelle Position der Zeile, die den Ausdruck „$“ enthält. Wir können also eine Endlosschleife codieren, wenn wir „JMP $“ schreiben.  Das Token „$$“ ist der Anfang des aktuellen Abschnitts. Anhand des Ausdrucks „($ - $$)“ können wir also feststellen, wie weit wir innerhalb des Abschnitts sind. |
| Formatul unei linii sursă |  | 1.2 Das Format einer Quellzeile |
| Formatul unei linii sursă în limbajul de asamblare x86 este următorul: |  | Das Format einer Quellzeile in der x86-Assemblersprache lautet wie folgt: |
| **[etichetă[:]] [prefixe] [mnemonică] [operanzi] [;comentariu]**  (***[Label [:]] [Präfixe] [Mnemonik] [Operanden] [; Kommentar]*** ) | | |
| Ilustrăm conceptul prin intermediul a câteva exemple de linii sursă: |  | Wir veranschaulichen das Konzept anhand einiger Beispiele für Quelltextzeilen: |
| *hier*: **jmp** *hier* ; haben wir Label + Mnemonik + operand + Kommentar  **repz** **cmpsd** ; Präfix + Mnemonik + Kommentar  *start*: ; Label + Kommentar  ; nur ein Kommentar (care putea lipsi și el - *was fehlen könnte*)  a dw 19872, 42h ; Label + Mnemonik + 2 Operanden + Kommentar | | |
| Caracterele din care poate fi constituită o *etichetă* sunt următoarele:  Litere, atât A-Z cât și a-z;  Cifre de la 0 la 9;  Caracterele \_, $, #, @, ~, . și ?  Ca prim caracter al unei etichete sunt permise doar litere, \_ și ?  Aceste reguli sunt valabile pentru toți *identificatorii* valizi (denumiri simbolice, precum nume de variabile, etichete, macro, etc.).  Identificatorii definiţi de utilizator sunt *case sensitive*: limbajul face distincţie între literele mari şi cele mici. Aceasta înseamnă că un identificator „Abc” este diferit de identificatorul „abc”. Pentru denumirile care fac implicit parte din limbaj, cum ar fi cuvintele cheie, mnemonicile și numele regiștrilor, nu se diferențiază literele mari de cele mici (acestea sunt *case insensitive*).  La nivelul limbajului de asamblare se întâlnesc două categorii de etichete:  1) ***etichete de cod***, care apar în cadrul secvenţelor de instrucţiuni (segmente de cod) cu scopul de a defini destinaţiile de transfer ale controlului în cadrul unui program.  2) ***etichete de date***, care identifică simbolic unele locaţii de memorie, din punct de vedere semantic ele fiind echivalentul noţiunii de *variabilă* din alte limbaje.  ***Valoarea* unei etichete în limbaj de asamblare este un număr întreg reprezentând adresa instrucţiunii, directivei sau datelor care urmează etichetei.**  Distincția dintre referirea adresei unei variabile sau a conținutului asociat acesteia se face după regulile:  Când este specificat ***între paranteze drepte, numele variabilei desemnează valoarea variabilei***, de exemplu [*p*] specifică accesarea valorii variabilei *p*, similar cu modul în care \**p* semnifică dereferențierea unui pointer (accesul la conținutul indicat prin valoarea pointerului) în C;  În orice alt context ***numele variabilei reprezintă adresa variabilei***, spre exemplu, *p* este întotdeauna adresa variabilei *p*;  *Exemple*:  **mov** EAX, et ; încarcă în registrul EAX **adresa** datelor sau a codului marcat cu eticheta „et”  **mov** EAX, [et]; încarcă în registrul EAX **conținutul** de la adresa „et” (4 octeţi)  **lea** EAX, [v] ; încarcă în registrul EAX **adresa** (*offset*-ul) variabilei „v” (4 octeţi)  ***Folosirea parantezelor pătrate indică întotdeauna accesarea unui operand din memorie***.  De exemplu, **mov** EAX, [EBX] semnifică un transfer în EAX a conţinutului memoriei a cărei adresă este dată de valoarea lui EBX.  Există două tipuri de *mnemonice*: mnemonice de *instrucţiuni* şi nume de *directive*. *Directivele* dirijează asamblorul. Ele specifică modul în care asamblorul va genera codul obiect. *Instrucţiunile* dirijează procesorul.  *Operanzii* sunt parametri care definesc valorile ce vor fi prelucrate de instrucţiuni sau de directive. Ei pot fi **regiştri, constante, etichete, expresii, cuvinte cheie sau alte simboluri**. Semnificaţia operanzilor depinde de mnemonica instrucţiunii sau directivei asociate. |  | Die Zeichen, aus denen ein Label gebildet werden kann, sind die folgenden:  Buchstaben von A bis Z und von a bis z;  Zahlen von 0 bis 9;  Die Zeichen \_, $, #, @, ~ ,. und?  Als erstes Zeichen eines Labels sind nur die Buchstaben \_ und? zulässig.  Diese Regeln gelten für alle gültigen *Bezeichner* (symbolische Namen wie Variablennamen, Bezeichnungen, Makros usw.). Benutzerdefinierte Bezeichner sind *case sensitive*: die Assemblersprache unterscheidt zwischen Groß- und Kleinschreibung. Dies bedeutet, dass sich eine „Abc“ Bezeichner von der „abc“ Bezeichner unterscheidet. Bei den Namen, die implizit Teil der Sprache sind, wie Schlüsselwörter, Mnemonik und Registernamen, unterscheiden sich Groß- und Kleinbuchstaben nicht (dies sind *case insensitive*). Auf Assemblersprachenebene gibt es zwei Kategorien von Labels: 1) ***Code-Labels***, die in den Anweisungssequenzen (Codesegmenten) erscheinen, um die Steuerübertragungsziele innerhalb eines Programms zu definieren. 2) ***Daten-Labels***, die einige Speicherorte symbolisch kennzeichnen, semantisch dem Begriff der *Variablen* in anderen Sprachen entsprechen.  **Der *Wert* eines Labels in Assemblersprache ist eine Ganzzahl, die die Adresse der Anweisung, Direktive oder Daten darstellt, die auf das Label folgen.**  Die Unterscheidung zwischen dem Verweis auf die Adresse einer Variablen oder ihrem Inhalt erfolgt nach folgenden Regeln:  Wenn in ***eckigen Klammern*** angegeben, ***gibt der Variablenname den Wert der Variablen an***. Beispielsweise gibt [*p*] den Zugriff auf den Wert der Variablen *p* an, ähnlich wie \**p* eine Zeiger-Dereferenzierung (Zugriff auf den durch den Zeigerwert angegebenen Inhalt) in C angibt.  In jedem anderen Kontext ***repräsentiert der Name der Variablen die Adresse der Variablen***, zum Beispiel ist *p* immer die Adresse der Variablen *p*;  *Beispiele*: **mov** EAX, et ; ladet in das EAX-Register die Daten**-** oder den Code**-adresse**, der mit dem Label „et“ markiert ist  **mov** EAX, [et] ; ladet den Inhalt von der Adresse „et“ (4 Bytes) in das EAX-Register **lea** EAX, [v]; ladet die Adresse (*Offset*) der Variablen „v“ (4 Bytes) in das EAX-Register  ***Die Verwendung von eckigen Klammern zeigt immer an, dass auf einen Operanden aus dem Speicher zugegriffen wird.***  Zum Beispiel bezeichnet **mov** EAX, [EBX] eine Übertragung des Speicherinhalts, dessen Adresse durch den Wert von EBX gegeben ist, an EAX.  Es gibt zwei Arten von *Mnemoniken*: Mnemoniken von Anweisungen und Namen von *Anweisungen*. Die *Direktiven* leiten den Assembler. Sie geben an, wie der Assembler den Objektcode generiert. Die *Anweisungen* leiten den Prozessor.  *Operanden* sind Parameter, die die Werte definieren, die von Anweisungen oder Direktiven verarbeitet werden. Dies können **Register, Konstanten, Bezeichnungen, Ausdrücke, Schlüsselwörter oder andere Symbole** sein. Die Bedeutung der Operanden hängt von der Mnemonik der zugehörigen Anweisung oder Direktive ab. |
| 2. Expresii |  | 2. Ausdrücke |
| O expresie este formată din operanzi şi operatori. Operatorii indică modul de combinare a operanzilor în scopul formării expresiei. Expresiile sunt evaluate în momentul asamblării (adică, valorile lor sunt determinabile la momentul asamblării, cu excepţia acelor părţi care desemnează conţinuturi de regiştri şi care vor fi determinate la execuţie). |  | Ein Ausdruck besteht aus Operanden und Operatoren. Die Operatoren geben an, wie die Operanden zum Ausdruck zusammengefasst werden. Die Ausdrücke werden zum Zeitpunkt des Assemblerung ausgewertet (dh ihre Werte sind zum Zeitpunkt des Assemblerung bestimmbar, mit Ausnahme derjenigen Teile, die den Inhalt von Registern bezeichnen und die bei der Ausführung bestimmt werden). |
| 2.1 Moduri de adresare |  | 2.1 Adressierungsarten |
| Operanzii instrucţiunilor pot fi specificaţi sub forme numite ***moduri de adresare***.  Cele trei tipuri de operanzi sunt: ***operanzi imediaţi, operanzi registru*** şi ***operanzi în memorie***.  Valoarea operanzilor este calculată în momentul asamblării pentru operanzii imediaţi, în momentul încărcării programului pentru adresarea directă (adresa FAR) şi în momentul execuţiei pentru operanzii registru şi cei adresaţi indirect. |  | Befehlsoperanden können in Form von ***Adressierungsmodi*** angegeben werden.  Die drei Arten von Operanden sind: ***Sofortoperanden***, ***Registeroperanden*** und ***Speicheroperanden***. Der Wert der Operanden wird zum Zeitpunkt des Assemblierungs für die unmittelbaren Operanden, zum Zeitpunkt des Ladens des Programms für die direkte Adressierung (FAR-Adresse) und zum Zeitpunkt der Ausführung für das Operandenregister und die indirekt adressierten Operanden berechnet. |
| 2.1.1 Utilizarea operanzilor imediaţi |  | 2.1.1 Verwendung von Sofortoperanden |
| *Operanzii imediaţi* sunt formaţi din date numerice constante calculabile la momentul asamblării.  Constantele sunt utilizate ca operanzi în expresii. Limbajul de asamblare recunoaşte patru tipuri de valori constante: **întregi, şiruri de caractere, numere reale** şi **constante împachetate codificate binar zecimal (BCD)**.  *Constantele întregi* se specifică prin valori binare, octale, zecimale sau hexazecimale. Adiţional, este permisă folosirea caracterului \_ (*underscore*) pentru a separa grupuri de cifre. Baza de numeraţie poate fi precizată în mai multe moduri:   * Folosind sufixele H sau X pentru hexazecimal, D sau T pentru zecimal, Q sau O pentru octal şi B sau Y pentru binar. În aceste cazuri, numărul trebuie neapărat să înceapă cu o cifră între 0 şi 9, pentru a nu exista confuzii între constante şi simboluri. De exemplu, 0ABCH este interpretat ca număr hexazecimal, dar ABCH este interpretat ca simbol. * În stilul specific limbajului C, prin prefixare cu 0x sau 0h pentru hexazecimal, 0d sau 0t pentru zecimal, 0o sau 0q pentru octal, respectiv 0b sau 0y pentru binar;   Exemple:   * constanta hexazecimală B2A poate fi exprimată ca 0xb2a, 0xb2A, 0hb2a, 0b12Ah, 0B12AH, etc; * valoarea zecimală 123 poate fi specificată ca 123, 0d123, 0d0123, 123d, 123D, ... * 11001000b, 0b11001000, 0y1100\_1000, 001100\_1000Y reprezintă diferite exprimări ale numărului binar 11001000.   O constantă de tip şir este formată din unul sau mai multe caractere ASCII delimitate de ghilimele sau de apostrofuri. Dacă printre aceste caractere ASCII trebuie să apară caracterul delimitator, acesta trebuie dublat. Exemple: ’a’, ”a”, ”Ia vino’ ’ncoa”.  Pentru unele instrucţiuni există o limită maximă în ceea ce priveşte dimensiunea de reprezentare a valorilor imediate (de obicei 8, 16 sau 32 de biţi). Constantele de tip şir care sunt mai lungi de două caractere (patru caractere la procesoarele 80386) nu pot fi date imediate. Ele trebuie să fie stocate în memorie înainte de a fi prelucrate de instrucţiuni.  Datele imediate nu sunt admise ca operand destinaţie (aşa cum nici 2:=N nu este o instrucţiune validă în nici un limbaj de programare de nivel înalt).  *Constantele întregi codificate zecimal* constituie un tip special de constante care pot fi utilizate numai pentru iniţializarea variabilelor BCD.  Cu *numere reale* se poate opera numai în prezenţa unui coprocesor matematic.  **Deplasamentele etichetelor de date şi de cod** reprezintă valori determinabile la momentul asamblării care rămân ***constante*** pe tot parcursul execuţiei programului.  De exemplu, o instrucţiune de genul:  **mov** EAX, et; transfer în registrul EAX a adresei efective asociate etichetei *et*  va putea fi evaluată la momentul asamblării drept, de exemplu,  **mov** eax, 8 ; distanţă de 8 octeţi faţă de începutul segmentului de date  „Constanţa” acestor valori derivă din regulile de alocare adoptate de limbajele de programare în general şi care statuează că ordinea de alocare în memorie a variabilelor declarate (mai precis distanţa faţă de începutul segmentului de date în care o variabilă este alocată) sau respectiv distanţele salturilor destinaţie în cazul unor instrucţiuni de tip **goto** sunt valori constante pe parcursul execuţiei unui program.  Adică: o variabilă odată alocată în cadrul unui segment de memorie nu îşi va schimba niciodată locul alocării (adică poziţia sa faţă de începutul acelui segment). Această informaţie poate fi determinată la momentul asamblării; ea derivă din ordinea specificării variabilelor la declarare în cadrul textului sursă şi din dimensiunea de reprezentare (dedusă pe bază informaţiei de tip asociate). |  | *Sofortige Operanden* bestehen aus dem konstanten numerischen Daten, das zum Zeitpunkt der Assemblierung berechnet werden kann.  Konstanten sind als Operanden in Ausdrücken verwendet. In der Assemblersprache werden vier Arten von Konstantenwerten erkannt: **Ganzzahlen**, **Zeichenfolgen**, **reelle** **Zahlen** und **binär kodierte Dezimalzahlen (BCD)**.  *Ganze Konstanten* sind spezifisch für Binär-, Oktal-, Dezimal- oder Hexadezimalwerte. Zusätzlich ist der Unterstrich von der Zeichen \_ (*underscore*) für eine separate Zifferngruppe zulässig. Die Zahlenbasis kann auf verschiedene Arten angegeben werden:   * Verwenden die Suffixen H oder X für hexadezimal, D oder T für dezimal, Q oder O für oktal und B oder Y für binär. In diesen Fällen muss die Zahl unbedingt mit einer Zahl zwischen 0 und 9 beginnen, damit Konstanten und Symbole nicht verwechselt werden. Beispielsweise wird 0ABCH als Hexadezimalzahl interpretiert, ABCH jedoch als Symbol. * In dem für die Sprache C spezifischen Stil mit dem Präfix 0x oder 0h für hexadezimal, 0d oder 0t für dezimal, 0o oder 0q für oktal bzw. 0b oder 0y für binär;   Beispiele:   * Die hexadezimale Konstante B2A kann ausgedrückt werden als 0xb2a, 0xb2A, 0hb2a, 0b12Ah, 0B12AH usw .; * Der Dezimalwert 123 kann als 123, 0d123, 0d0123, 123d, 123D, ... angegeben werden; * 11001000b, 0b11001000, 0y1100\_1000, 001100\_1000Y stellen unterschiedliche Ausdrücke der Binärzahl 11001000 dar.   Eine Zeichenfolgekonstante besteht aus einem oder mehreren ASCII-Zeichen, die durch Anführungszeichen oder Apostrophe getrennt sind. Wenn das Trennzeichen zwischen diesen ASCII-Zeichen stehen muss, muss es verdoppelt werden. Beispiele: ’a’, ”a”, ”Ia vino’ ’ncoa”.  Für einige Anweisungen gibt es eine maximale Grenze für die Größe der Sofortwertdarstellung (normalerweise 8, 16 oder 32 Bit). Zeichenfolgentypkonstanten, die länger als zwei Zeichen sind (vier Zeichen bei 80386-Prozessoren), können nicht sofortige Daten werden. Sie müssen gespeichert werden, bevor sie von den Anweisungen verarbeitet werden.  Sofortige Daten sind als Zieloperand nicht zulässig (da entweder 2: = N keine gültige Anweisung in einer höheren Programmiersprache ist).  *Dezimal codierte Ganzzahlkonstanten* sind eine spezielle Art von Konstanten, die nur zum Initialisieren von BCD-Variablen verwendet werden können.  Mit *reellen Zahlen* können Sie nur in Gegenwart eines mathematischen Coprozessors arbeiten.  **Die Offseten der Daten- und Code-Label** stellen bestimmbare Werte zum Zeitpunkt der Zusammenstellung dar, die während der Ausführung des Programms ***konstant*** bleiben. Zum Beispiel eine Anweisung wie:  **mov** EAX, et; Übertragen der dem Label *et* zugeordneten tatsächlichen Adresse in das EAX-Register  kann zum Zeitpunkt der Assemblierung ausgewertet werden als, zu Beispiel:  **mov** eax, 8; 8 Byte Abstand vom Beginn des Datensegments  Die „Konstanz“ dieser Werte ergibt sich aus den von den Programmiersprachen allgemein übernommenen Zuordnungsregeln und der Reihenfolge der Zuordnung der deklarierten Variablen (genauer der Abstand vom Beginn des Datensegments, in dem eine Variable zugeordnet ist) bzw. den Sprungdistanzen im Speicher Ziel bei **goto**-Anweisungen sind konstante Werte während der Ausführung eines Programms.  Das heißt: Eine Variable, die einmal innerhalb eines Speichersegments zugewiesen wurde, ändert niemals ihren Zuweisungsort (deshalb ihre Position relativ zum Beginn dieses Segments). Diese Informationen können zum Zeitpunkt der Assemblierung ermittelt werden; sie ergibt sich aus der Reihenfolge der Angabe der zu deklarierenden Variablen im Quelltext und aus der Dimension der Darstellung (abgeleitet aus den zugehörigen Typinformationen). |
| 2.1.2 Utilizarea operanzilor registru |  | 2.1.2 Verwendung von Registrierungsoperanden |
| Modul de *adresare directă*:  **mov** EAX, EBX; provoacă transferul valorii din registrul BX în registrul AX.  *Adresare* *indirectă*,pentru a indica locaţiile de memorie:  **mov** EAX, [EBX]; provoacă transferul cuvântului de memorie de la adresa desemnată de registrul BX spre registrul AX. |  | *Direktadressierungsmodus*:  **mov** EAX, EBX; bewirkt die Übertragung des Wertes vom EBX-Register in das EAX-Register.  *Indirekte Adressierung* zur Angabe von Speicherplätzen: **mov** EAX, [EBX]; Bewirkt die Wortübertragung von der vom EBX-Register angegebenen Adresse zum EAX-Register. |
| 2.1.3 Utilizarea operanzilor din memorie |  | 2.1.3 Verwenden der Operanden aus dem Speicher |
| Operanzii din memorie sunt cu *adresare directă* şi cu *adresare indirectă*.  Operandul cu *adresare directă* este o constantă sau un simbol care reprezintă adresa (segment şi deplasament) unei instrucţiuni sau a unor date. Aceşti operanzi pot fi *etichete* (de ex: **jmp** et), *nume de proceduri* (de ex: **call** proc1) sau *valoarea contorului de locaţii* (de ex: b db $-a).  ***Deplasamentul unui operand cu adresare directă este calculat în momentul asamblării*** (adică la „*assembly time*”). Adresa fiecărui operand raportată la structura programului executabil (mai precis stabilirea segmentelor la care se raportează deplasamentele calculate) este calculată în momentul editării de legături (*linking time*). Adresa fizică efectivă este calculată în momentul încărcării programului pentru execuţie (*loading time*).  Adresa efectivă este întotdeauna raportată la un registru de segment. Acest registru poate fi specificat explicit sau, în caz contrar, se asociază de către asamblor în mod implicit un registru de segment. Regulile pentru asocierile implicite sunt:  **CS** pentru etichete de cod destinaţie ale unor salturi (**jmp**, **call**, **ret**, **jz** etc);  **SS** în adresări SIB ce foloseşte **EBP** sau **ESP** drept bază (indiferent de index sau scală);  **DS** pentru restul accesărilor de date.  Specificarea explicită a unui registru de segment se face cu ajutorul operatorului de prefixare segment (notat „**:**” şi care se mai numeşte, 'operatorul de specificare a segmentului') .  Observaţie: Dacă este omisă eticheta din adresarea directă folosită cu un index constant (de exemplu omiterea etichetei *table* din exprimarea *table*[100h]), este necesară atunci specificarea unui segment. Deplasamentul operandului este considerat drept punctul de început al segmentului specificat (care trebuie să aibă aceeaşi valoare cu deplasamentul etichetei *table* în cazul nostru) plus deplasamentul indexat. De exemplu, DS:[100h] reprezintă valoarea de la adresa 100h din segmentul referit de DS, exprimare echivalentă cu DS:100h.  Dacă se omite specificarea segmentului, este folosită valoarea constantă (imediată) a operandului şi nu valoarea pe care o indică. De exemplu, [100h] desemnează chiar valoarea 100h, şi nu valoarea de la adresa 100h.  Cât priveşte *operanzii cu adresare indirectă*, aceştia utilizează regiştri pentru a indica adrese din memorie. Deoarece valorile din regiştri se pot modifica la momentul execuţiei, adresarea indirectă este indicată pentru a opera în mod dinamic asupra datelor.  Forma generală pentru accesarea indirectă a unui operand de memorie este dată de formula de calcul a *offset*-ului unui operand: |  | Die Operanden im Speicher haben eine *direkte* und eine *indirekte* *Adressierung*.  Der *direkte Adressierungsoperand* ist eine Konstante oder ein Symbol, das die Adresse (Segment und Verschiebung) eines Anweisungs oder von Daten darstellt. Diese Operanden können *Labels* (zB: **jmp** et), *Prozedurnamen* (zB: **call** proc1) oder *der Wert des Programmzählers* (zB: b db $-a) sein. ***Das Offset eines direkten Adressoperanden wird zum Zeitpunkt des Assemblierungs*** (dh zum „*assembly time*”) ***berechnet***. Die Adresse jedes Operanden, die sich auf die Struktur des ausführbaren Programms bezieht (genauer gesagt, die Festlegung der Segmente, auf die sich die berechneten *Offseten* beziehen), wird zum Zeitpunkt der Verknüpfung (*linking time*) berechnet. Die tatsächliche physikalische Adresse wird beim Laden des Programms zur Ausführung berechnet (*loading time*). Die tatsächliche Adresse wird immer an ein Segmentregister gemeldet. Dieses Register kann explizit angegeben werden, andernfalls wird das Segment standardmäßig einem Segmentregister zugeordnet. Die Regeln für die Standardzuordnungen sind:  **CS** für Zielcode-Labels einiger Sprünge (**jmp**, **call**, **ret**, **jz** usw.);  **SS** in SIB-Adressen, die **EBP** oder **ESP** als Basis verwenden (unabhängig von Index oder Skala);  **DS** für den Rest der Datenzugriffe.  Die explizite Angabe eines Segmentregisters erfolgt mit Hilfe des Segmentpräfix-Operators (vermerkt mit „:“ und der auch als "Segmentspezifikationsoperator" bezeichnet wird).  Hinweis: Wenn das Label in der direkten Adresse, die mit einem konstanten Index verwendet wird, weggelassen wird (z. B. die Label *table* in der Ausdruck *table*[100h] weggelassen wird), muss ein Segment angegeben werden. Das *Offset* des Operands wird als Startpunkt des angegebenen Segments (das in unserem Fall den gleichen Wert wie das *Offset* des Label *table* haben muss) plus der indizierten *Offset* betrachtet. Zum Beispiel stellt DS:[100h] den Wert an der Adresse 100h des Segments dar, auf das sich DS bezieht, ein Ausdruck, der DS:100h entspricht. Wenn die Segmentangabe weggelassen wird, wird der konstante (unmittelbare) Wert des Operanden verwendet und nicht der angegebene Wert. Beispielsweise bezeichnet [100h] sogar den Wert 100h und nicht den Wert an der Adresse 100h.  Bei *den Operanden für indirekte Adressen* verwenden sie Register, um Adressen aus dem Speicher anzuzeigen. Da die Werte in den Registern zur Laufzeit geändert werden können, wird die indirekte Adressierung angegeben, um die Daten dynamisch zu verarbeiten.  Die allgemeine Form für den indirekten Zugriff auf einen Speicheroperanden ergibt sich aus der Formel zur Berechnung des *Offsets* eines Operanden: |
| [ registru\_de\_bază ] + [ registru\_index × scală ] + [ constantă ]  ( *[Basis\_Register] + [Index\_Register × Skala] + [Konstante]* ) | | |
| Constanta este o expresie a cărei valoare este determinabilă la momentul asamblării. De exemplu, [EBX + EDI + *table* + 6] desemnează un operand prin adresare indirectă, unde atât *table* cât şi 6 sunt constante.    Operanzii *registru\_de\_bază* şi *registru\_index* sunt folosiţi de obicei pentru a indica o adresă de memorie referitoare la un tablou. În combinaţie cu factorul de scalare, mecanismul este suficient de flexibil pentru a permite acces direct la elementele unui tablou de înregistrări, cu condiţia ca dimensiunea în octeţi a unei înregistrări să fie 1, 2, 4 sau 8. De exemplu, octetul superior al elementului de tip DWORD cu index dat în **ECX**, parte a unui vector de înregistrări al cărui adresă (a vectorului) este în **EDX** poate fi încărcat în **DH** prin intermediul instrucţiunii  **mov** DH, [EDX + ECX \* 4 + 3]  Din punct de vedere sintactic, când operandul nu este specificat prin formula completă (deoarece lipsesc unele dintre componente – de exemplu lipseşte partea „\* scală”), atunci asamblorul va rezolva ambiguitatea care rezultă analizând toate formele echivalente de codificare posibile şi alegând-o pe cea mai scurtă dintre acestea. Cu alte cuvinte, având  **push** dword [EAX+EBX]; salvează pe stivă dublucuvântul de la adresa EAX + EBX  asamblorul are libertatea de a considera EAX drept bază şi EBX drept index sau invers, EBX drept bază şi EAX drept index.  Analog, pentru  **pop** dword [ECX]; restaurează vârful stivei în variabila cu adresa dată de ECX  asamblorul poate interpreta ECX fie ca bază fie ca index. Ce este realmente important de reţinut este faptul că toate codificările luate în considerare de către asamblor sunt echivalente iar decizia finală a asamblorului nu are impact asupra funcţionalităţii codului rezultat.  De asemenea, în plus faţă de rezolvarea unor astfel de ambiguităţi, asamblorul permite şi exprimări non-standard cu condiţia ca acestea să fie transformabile într-un final în forma standard de mai sus.  Alte exemple:  **lea** EAX, [EAX\*2]; încarcă în EAX valoarea lui EAX\*2 (adică,  EAX devine 2\*EAX)  În acest caz, asamblorul poate decide între codificare de tip bază = EAX + index = EAX şi scală = 1 sau index = EAX şi scală = 2.  **lea** EAX, [EAX\*9 + 12]; EAX ia valoarea EAX \* 9 + 12  Deşi scală nu poate fi 9, asamblorul nu va emite aici un mesaj de eroare. Aceasta deoarece el va observa posibila codificare a adresei drept: bază = EAX + index = EAX cu scală = 8, unde de această dată valoarea 8 este corectă pentru scală. Evident, instrucţiunea putea fi precizată mai clar sub forma următoare: |  | Die Konstante ist ein Ausdruck, dessen Wert zum Zeitpunkt der Assemblierung bestimmbar ist. Beispielsweise bezeichnet [EBX + EDI + *table* + 6] einen Operanden durch indirekte Adressierung, wobei beide *table* und 6 konstant sind.  Die Operanden *Basis\_Register* und *Index\_Register* werden normalerweise verwendet, um eine Speicheradresse für eine Tabelle anzugeben. In Kombination mit dem Skalierungsfaktor ist der Mechanismus flexibel genug, um einen direkten Zugriff auf die Elemente eines Aufzeichnungstabelles zu ermöglichen, vorausgesetzt, die Bytegröße einer Aufzeichnung beträgt 1, 2, 4 oder 8. Zum Beispiel das obere Byte des Elements vom Typ DWORD mit in **ECX** angegebenem Index, Teil eines Datensatzvektors, dessen Adresse (des Vektors) in **EDX** ist, kann über die Anweisung  **mov** DH, [EDX + ECX \* 4 + 3]  in **DH** hochgeladen werden.  Aus syntaktischer Sicht löst der Assembler die Mehrdeutigkeit, die sich aus einem Analyseprozess aller möglichen äquivalenten Codierungsformen und ergibt, auf, wenn der Operand nicht durch die vollständige Formel angegeben wird und einige der Komponenten fehlen (z. B. gibt es keine "\* Skala") Wählen Sie die kürzeste davon. Mit anderen Worten, mit  **push** dword [EAX + EBX]; Speichert das Doppelwort von der Adresse EAX + EBX auf dem Stapel  der Assembler hat die Freiheit, EAX als Basis und EBX als Index oder umgekehrt, EBX als Basis und EAX als Index zu betrachten.  Analog, ist  **pop** Dword [ECX]; Stellt die Stapelspitze in der Variablen mit der von ECX angegebenen Adresse wieder her  der Assembler kann ECX entweder als Basis oder als Index interpretieren. Es ist wirklich wichtig, sich daran zu erinnern, dass alle vom Assembler berücksichtigten Codierungen gleichwertig sind und die endgültige Entscheidung des Assemblers keinen Einfluss auf die Funktionalität des resultierenden Codes hat.  Zusätzlich zum Auflösen solcher Mehrdeutigkeiten erlaubt der Assembler auch nicht standardmäßige Ausdrücke, vorausgesetzt, sie können endgültig in die obige Standardform umgewandelt werden.  Andere Beispiele:  **lea** EAX, [EAX \* 2]; Ladet in EAX den Wert von EAX\*2 (dh, EAX wird 2\*EAX)  In diesem Fall kann der Assembler zwischen Basistypcodierung = EAX + Index = EAX und Skala = 1 oder Index = EAX und Skala = 2 wählen.  **lea** EAX, [EAX \* 9 + 12]; EAX nimmt den Wert EAX \* 9 + 12 an  Obwohl die Skala nicht 9 sein kann, gibt der Assembler hier keine Fehlermeldung aus. Dies liegt daran, dass er die mögliche Kodierung der Adresse als: base = EAX + index = EAX mit scale = 8 bemerkt, wobei diesmal der Wert 8 für die Skala korrekt ist. Offensichtlich könnte die Anweisung in der folgende Form klarer spezifiziert werden |
| **lea** EAX, [EAX + EAX \* 8 + 12] | | |
| Să reţinem deci că pentru adresarea indirectă, esenţială este specificarea între paranteze drepte a cel puţin unuia dintre elementele componente ale formulei de calcul a *offset*-ului. |  | Daher ist zu beachten, dass für die indirekte Adressierung mindestens eines der Komponentenelemente der Offsetberechnungsformel in Klammern angegeben werden muss. |
| 2.2 Utilizarea operatorilor |  | 2.2 Verwendung von Operatoren |
| Operatorii se folosesc pentru combinarea, compararea, modificarea şi analiza operanzilor. Unii operatori lucrează cu constante întregi, alţii cu valori întregi memorate, iar alţii cu ambele tipuri de operanzi.  Este importantă înţelegerea diferenţei dintre operatori şi instrucţiuni. Operatorii efectuează calcule cu valori constante determinabile la momentul asamblării. Instrucţiunile efectuează calcule cu valori ce pot fi necunoscute până în momentul execuţiei. Operatorul de adunare (+) efectuează adunarea în momentul asamblării; instrucţiunea ADD efectuează adunarea în timpul execuţiei.    Operatorii disponibili pentru construcţia expresiilor sunt asemănători celor din limbajul C, atât ca sintaxă cât şi din punct de vedere semantic. Evaluarea expresiilor numerice se face pe 64 de biţi, rezultatele finale fiind ulterior ajustate în conformitate cu dimensiunea de reprezentare disponibilă în contextul de utilizare al expresiei.  În Tabelul 1 sunt prezentaţi în ordinea priorităţii operatorii care pot fi folosiţi în cadrul expresiilor limbajului de asamblare x86. |  | Operatoren werden zum Kombinieren, Vergleichen, Ändern und Analysieren von Operanden verwendet. Einige Operatoren arbeiten mit Ganzzahlkonstanten, andere mit gespeicherten Ganzzahlwerten und andere mit beiden Arten von Operanden.  Es ist wichtig, den Unterschied zwischen Operanden und Anweisungen zu verstehen. Die Operatoren führen Berechnungen mit konstanten Werten durch, die zum Zeitpunkt der Assemblierung ermittelt werden können. Die Anweisungen führen Berechnungen mit Werten durch, die bis zum Zeitpunkt der Ausführung unbekannt sein können. Der Additionoperator (+) führt die zum Zeitpunkt der Assemblierung durch. Die ADD-Anweisung führt die Addition während der Ausführung durch.  Die zur Erstellung von Ausdrücken verfügbaren Operatoren sind syntaktisch und semantisch denen in C-Sprache ähnlich. Die Auswertung der numerischen Ausdrücke erfolgt mit 64 Bit, wobei die Endergebnisse anschließend entsprechend der im Zusammenhang mit der Verwendung des Ausdrucks verfügbaren Darstellungsgröße angepasst werden.  In Tabelle 1 sind die Operatoren, die in den Ausdrücken der Assemblersprache x86 verwendet werden können, in der Reihenfolge ihrer Priorität aufgeführt. |
| **Tabel 1.** *Operatorii care pot fi folosiţi în cadrul expresiilor limbajului de asamblare x86.*   |  |  |  |  | | --- | --- | --- | --- | | **Prioritate** | **Operator** | **Tip** | **Rezultat** | | 7 | - | Unar, präfix | Zweier Komplement (Negation): -X = 0 – X | | 7 | + | Unar, präfix | Kein Effekt (angeboten für Symmetrie mit „-“): +X = X | | 7 | ~ | Unar, präfix | Einer Komplement 1: **mov** AL, ~0 => **mov** AL, 0xFF | | 7 | ! | Unar, präfix | Logische Negation: !X = 0 wenn X = 0, andernfalls 1 | | 6 | \* | Binar, infix | Multiplikation: 1 \* 2 \* 3 = 6 | | 6 | / | Binar, infix | Der Betrag der Division ohne Vorzeichen: 24 / 4 / 2 = 3 (-24/4/2 = 0FDh) | | 6 | // | Binar, infix | Der Betrag der Division mit Vorzeichen: -24 // 4 // 2 = -3 ***(-24 / 4 / 2 ≠ -3!)*** | | 6 | % | Binar, infix | Der Rest der Division ohne Vorzeichen: 123 % 100 % 5 = 3 | | 6 | %% | Binar, infix | Der Rest der Division mit Vorzeichen: -123 %% 100 %% 5 = -3 | | 5 | + | Binar, infix | Aufsummierung: 1 + 2 = 3 | | 5 | - | Binar, infix | Abnehmen: 1 – 2 = -1 | | 4 | << | Binar, infix | Bitweise Linksverschiebung: 1 << 4 = 16 | | 4 | >> | Binar, infix | Bitweise Rechtsverschiebung: 0xFE >> 4 = 0x0F | | 3 | & | Binar, infix | UND: 0xF00F & 0x0FF6 = 0x0006 | | 2 | ^ | Binar, infix | Exklusiv ODER: 0xFF0F ^ 0xF0FF = 0x0FF0 | | 1 | | | Binar, infix | ODER: 1 | 2 = 3 | | | |
| Operatorul de indexare are o utilizare largă în specificarea operanzilor din memorie adresaţi indirect. Rolul operatorului [] în adresarea indirectă a fost clarificat anterior. |  | Der Indexierungsoperator wird häufig bei der Angabe von indirekt adressierten Speicheroperanden verwendet. Die Rolle des Operators [] bei der indirekten Adressierung wurde bereits erläutert. |
| 2.2.1 Operatori de deplasare de biţi |  | 2.2.1 Bitweise Verschiebungoperatoren |
| *expresie >> cu\_cât* şi *expresie << cu\_cât*  Exemple:  **mov** AX, 01110111b << 3; desemnează valoarea 10111000b  **add** BX, 01110111b >> 3 ; desemnează valoarea 00001110b |  | *Ausdruck >> wie viel* und *Ausdruck << wie viel* Beispiele: **mov** AX, 01110111b << 3; bezeichnet den Wert 10111000b  **add** BX, 01110111b >> 3; bezeichnet den Wert 00001110b |
| 2.2.2 Operatori logici pe biţi |  | 2.2.2 Bitweise Logischeoperatoren |
| *Operatorii pe biţi* efectuează operaţii logice la nivelul fiecărui bit al operandului (operanzilor) unei expresii. Expresiile au ca rezultat valori constante. |  | *Die Bitweise Operatoren* führen logische Operationen auf der Ebene jedes Bits des Operanden eines Ausdrucks aus. Ausdrücke führen zu konstanten Werten. |
| |  |  |  | | --- | --- | --- | | **OPERATOR** | **SYNTAX** | **SEMNIFICAŢIE (*BEDEUTUNG*)** | | **~** | ~ expresie | Bitweise Verneinung Ausdruck | | & | expr1 & expr2 | Bitweise UND | | | | expr1 | expr2 | Bitweise ODER | | ^ | expr1 ^ expr2 | Bitweise ODER EXKLUSIV | | | |
| Exemple (presupunem că expresia se reprezintă pe un octet):  ~11110000b; rezultă valoarea 00001111b  01010101b & 11110000b; rezultă valoarea 01010000b  01010101b | 11110000b; rezultă valoarea 11110101b  01010101b ^ 11110000b; rezultă valoarea 10100101b  ! – negare logică (similar cu limbajul C) ; !0 = 1 ; !(orice diferit de zero) = 0 |  | Beispiele (angenommen, der Ausdruck ist ein Byte):  ~ 11110000B; Der Wert 00001111b ergibt sich 01010101b & 11110000b; es ergibt sich der Wert 01010000b  01010101b | 11110000B; es ergibt sich der Wert 11110101b  01010101b ^ 11110000b; es ergibt sich der Wert 10100101b  ! - logische Verneinung (ähnlich der Sprache C); ! 0 = 1; ! (etwas anderes als Null) = 0 |
| 2.2.3 Operatorul de specificare a segmentului |  | 2.2.3 Der Segmentspezifikationsoperator |
| *Operatorul de specificare a segmentului* (**:**) comandă calcularea adresei FAR a unei variabile sau etichete în funcţie de un anumit segment. Sintaxa este:  ***segment*:*expresie***  [SS: EBX+4]; deplasamentul e relativ la SS  [ES:082h] ; deplasamentul e relativ la ES  10h:var ; segmentul este indicat de selectorul 10h, iar *offset*-ul este valoarea etichetei *var*. |  | Der Segmentspezifikationsoperator (:) befiehlt die FAR-Adressberechnung einer Variablen oder eines Labels basierend auf einem bestimmten Segment. Die Syntax lautet:                 ***Segment: Ausdruck***  [SS:EBX+4]; Verschiebung ist relativ zu SS [ES:082h]; Verschiebung ist relativ zu ES 10h:var; Das Segment wird durch den Selektor 10h angezeigt, und der *Offset* ist der Wert des var-Labels. |
| 2.2.4 Operatori de tip |  | 2.2.4 Typoperatoren |
| Specifică tipurile unor expresii şi a unor operanzi păstraţi în memorie. Sintaxa pentru aceştia este: |  | Typoperatoren Sie geben die im Speicher abgelegten Arten von Ausdrücken und Operanden an. Die Syntax für sie lautet: |
| ***tip* *expresie***  ***(Typ Ausdruck)*** | | |
| Această construcţie sintactică forţează ca *expresia* să fie tratată considerându-se că are dimensiunea de reprezentare 10 *tip*, însă fără ca valoarea sa să fie modificată în mod definitiv (distructiv) în sensul precizat de conversia dorită. De aceea, aceştia sunt consideraţi *operatori de conversie (temporară) nedistructivă*. Pentru operanzii păstraţi în memorie, *tip* poate fi **BYTE**, **WORD**, **DWORD, QWORD** sau **TWORD** având dimensiunile de reprezentare 1, 2, 4, 8 şi respectiv 10 octeţi. Pentru etichetele de cod el poate fi **NEAR** (adresă pe 4 octeţi) sau **FAR** (adresă pe 6 octeţi).    Expresia **byte [**A] va indica doar primul octet de la adresa indicată de A. Analog, **dword [**A] indică dublucuvântul ce începe la adresa A.  Specificatorii BYTE / WORD / DWORD / QWORD au întotdeauna doar rol de a clarifica o ambiguitate (inclusiv când este vorba despre o variabilă de memorie, faptul de a preciza **mov** BYTE [v], 0 sau **mov** WORD [v], 0 este tot o clarificare a ambiguităţii, întrucât nasm nu asociază faptul ca v este byte/ word / dword).  În cazul instrucţiunii  **mov** [v], 0  se va primi mesajul „syntax error – operation size not specified”!  **Specificatorul QWORD nu intervine niciodată explicit** în cod pe 32 de biţi.  Exemple unde e necesar un specificator de dimensiune al operanzilor:  - **mov** [mem], 12  - **(i)div** [mem] ; **(i)mul** [mem]  - **push** [mem] ; **pop** [mem]  - **push** 15 – aici este o inconsistenţă în NASM, asamblorul nu va emite eroare/warning ci va face **push** DWORD 15  Exemple de operanzi IMPLICIŢI efectiv pe 64 biţi (în cod pe 32):  **mul** dword [v]; înmulţeşte EAX cu dword-ul de la adresa v şi depune rezultatul în perechea de registre EDX:EAX  **div** dword [v]; împărţire EDX:EAX la v |  | Diese syntaktische Konstruktion erzwingt die Behandlung des *Ausdrucks* unter Berücksichtigung der Typdarstellungsdimension *Typ*, ohne dass sein Wert in dem durch die gewünschte Konvertierung angegebenen Sinne endgültig modifiziert (destruktiv) wird. Sie gelten daher als *nicht destruktive (temporäre) Konvertierungsoperatoren*. Für im Speicher abgelegte Operanden kann der *Typ* **BYTE**, **WORD**, **DWORD**, **QWORD** oder **TWORD** mit den Darstellungsgrößen 1, 2, 4, 8 bzw. 10 Byte sein. Für Code-Labels kann **NEAR** (4-Byte-Adresse) oder **FAR** (6-Byte-Adresse) angegeben werden.  Das Ausdruck **byte** [A] gibt nur das erste Byte der durch A angegebenen Adresse an. Analog gibt **dword** [A] das Doppelwort an, das mit der Adresse A beginnt.  BYTE / WORD / DWORD / QWORD-Spezifizierer haben immer nur die Rolle, eine Mehrdeutigkeit zu klären (einschließlich der Angabe von **mov** BYTE [v], 0 oder **mov** WORD [v], 0 ist alles a Klarstellung der Mehrdeutigkeit, da nasm nicht die Tatsache assoziiert, dass v Byte / Wort / Dwort ist).  Im Falle von Anweisungen  **mov** [v], 0  Die Meldung „Syntaxfehler – Operationsgröße nicht angegeben“ („syntax error – operation size not specified”!) wird empfangen! **Der QWORD-Bezeichner greift niemals explizit** in 32-Bit-Code **ein**.  Beispiele, bei denen ein Operandengrößenbezeichner erforderlich ist:  - **mov** [mem], 12  - **(i)div** [mem]; **(i)mul** [mem]  - **push** [mem]; **pop** [mem]  - **push** 15 – Hier liegt eine Inkonsistenz in NASM vor. Die Assembler gibt keinen Fehler / keine Warnung aus, jedoch **push** DWORD 15  Beispiele für die impliziter Operanden 64-Bit effektiv (in 32-Bit-Code):  **mul** dword [v]; multiplizieren Sie EAX mit dem dword von Adresse v und senden Sie das Ergebnis in das EDX:EAX Registerparr **div** dword [v]; Division des EDX:EAX durch v |
| 3. Directive |  | 3. Direktiven |
| Directivele indică modul în care sunt generate codul şi datele în momentul asamblării. |  | Die Direktiven geben an, wie der Code und die Daten zum Zeitpunkt der Assemblierung generiert werden. |
| 3.1 Directiva SEGMENT |  | 3.1 Die SEGMENT- Direktive |
| Directiva SEGMENT permite direcţionarea octeţilor de cod sau date emişi de către un asamblor înspre segmentul precizat, segment care poartă un nume şi are asociate diverse caracteristici. |  | Die SEGMENT-Direktive ermöglicht die Weiterleitung von Code- oder Datenbytes, die von einem Assembler ausgegeben werden, an das angegebene Segment, ein Segment, das einen Namen trägt und mit dem verschiedene Eigenschaften verknüpft sind. |
| **SEGMENT** nume [tip] [ALIGN = aliniere] [combinare] [utilizare] [CLASS = clasă]  (**SEGMENT** Name [Typ] [ALIGN = Ausrichtung] [Kombination] [Verwendung] [CLASS = Klasse]) | | |
| Numelui segmentului i se asociază ca valoare adresa de segment (32 biţi) corespunzătoare poziţiei segmentului în memorie în faza de execuţie. În acest sens, asamblorul NASM pune la dispoziţie şi simbolul special $$ care este echivalent cu adresa segmentului curent, acesta având însă avantajul că poate fi utilizat în orice context, fără a fi necesar să fie cunoscut numele segmentului în care ne aflăm.  Cu excepţia numelui, toate celelalte câmpuri sunt opţionale din punct de vedere atât al prezenţei, cât şi al ordinii în care sunt specificate.  Argumentele opţionale *tip*, *aliniere, combinare, utilizare* şi *'clasa'* dau editorului de legături şi asamblorului indicaţii referitoare la modul de încărcare şi atributele segmentelor.  Tip permite selectarea unui model de folosire a segmentului, având la dispoziție următoarele opțiuni:   * **code** (sau **text**) – segmentul va conține cod, conținutul nu poate fi scris dar se poate citi sau executa; * **data** (sau **bss**) – segment de date permițând citire și scriere, însă nu și execuție (valoare implicită); * **rdata** - segment din care se poate doar citi, menit a conține definiții de date constante   Argumentul opţional *aliniere* specifică multiplul numărului de octeţi la care trebuie să înceapă segmentul respectiv. Alinierile acceptate sunt puteri ale lui 2, între 1 și 4096.  Dacă argumentul *aliniere* lipseşte, atunci se consideră implicit că este vorba despre o aliniere ALIGN = 1, adică segmentul poate începe la orice adresă.  Argumentul opţional *combinare* controlează modul în care segmente cu acelaşi nume din cadrul altor module vor fi combinate cu segmentul în cauză la momentul editării de legături. Valorile posibile sunt:   * **PUBLIC** – indică editorului de legături să concateneze acest segment cu alte eventuale segmente cu acelaşi nume, obţinându-se un unic segment a cărui lungime este suma lungimilor segmentelor componente. * **COMMON** – specifică faptul că începutul acestui segment trebuie să se suprapună peste începutul tuturor segmentelor care au acelaşi nume. Se obţine un segment având dimensiunea egală cu cea a celui mai mare segment având acelaşi nume. * **PRIVATE –** indică editorului de legături că acest segment nu este permis a fi combinat cu altele care poartă același nume. * **STACK** – segmentele cu acelaşi nume vor fi concatenate. În faza de execuţie segmentul rezultat va fi segmentul stivă.   Implicit, dacă nu se specifică o metodă de combinare, orice segment este considerat PUBLIC.  Argumentul *utilizare* (*use*) permite optarea pentru altă dimensiune de cuvânt decât cea de 16 biți, care este implicită in lipsa precizării acestui argument.  Argumentul *'clasa'* are rolul de a permite stabilirea ordinii în care editorul de legături plasează segmentele în memorie. Toate segmentele având aceeaşi clasă vor fi plasate într-un bloc contiguu de memorie indiferent de ordinea lor în cadrul codului sursă. Nu există o valoare implicită de iniţializare pentru acest argument. Dacă nu este specificat, nu este definit. Drept consecinţă, se evită concatenarea într-un bloc continuu a tuturor segmentelor definite astfel. |  | Der Segmentname ist der Segmentadresse (32 Bit) zugeordnet, die der Position des Segments im Speicher in der Ausführungsphase entspricht. In dieser Hinsicht stellt die NASM-Assembler auch das spezielle Symbol $$ bereit, das ist äquivalent zu der aktuellen Segmentadresse. Dies hat jedoch den Vorteil, dass es in jedem Kontext verwendet werden kann, ohne den Namen des Segments zu kennen, in dem wir uns befinden.  Mit Ausnahme des Namens sind alle anderen Felder in Bezug auf Vorhandensein und Reihenfolge, in der sie angegeben sind, optional.  Optionale Argumente für *Typ*, *Ausrichtung*, *Kombination*, *Verwendung* und *Klasse* geben dem Link-Editor und dem Assembler Anweisungen zum Laden und Segmentieren von Attributen.  Mit Typ können wir ein Modell zur Verwendung des Segments auswählen. Folgende Optionen stehen zur Verfügung:   * **Code** (oder **Text**) – Das Segment enthält Code. Der Inhalt kann nicht geschrieben, aber gelesen oder ausgeführt werden; * **data** (oder **bss**) – Datensegment zum Lesen und Schreiben, jedoch nicht zur Ausführung (Standard); * **rdata** – Segment, aus dem Sie nur lesen können und das konstante Datendefinitionen enthalten soll.   Das optionale *Ausrichtungsargument* gibt das Vielfache der Anzahl der Bytes an, wo das Segment beginnen muss. Akzeptierte Ausrichtungen sind Zweierpotenzen zwischen 1 und 4096.  Wenn das *Ausrichtungsargument* fehlt, wird es standardmäßig als Alignment ALIGN = 1 betrachtet, dh das Segment kann an einer beliebigen Adresse beginnen.  Das optionale *Kombinationsargument* steuert, wie Segmente mit demselben Namen in anderen Modulen mit dem betreffenden Segment kombiniert werden, wenn Links bearbeitet werden. Mögliche Werte sind:   * **PUBLIC** – gibt dem Linkeditor an, dieses Segment mit anderen möglichen Segmenten desselben Namens zu verketten, um ein einzelnes Segment zu erhalten, dessen Länge die Summe der Längen der Komponentensegmente ist. * **COMMON** – gibt an, dass der Anfang dieses Segments den Anfang aller gleichnamigen Segmente überlappen muss. Man erhält ein Segment mit der Größe des größten gleichnamigen Segments. * **PRIVATE** – zeigt dem Linkeditor an, dass dieses Segment nicht mit anderen Segmenten mit demselben Namen kombiniert werden darf. * **STACK** – Segmente mit demselben Namen verkettet werden. In der Ausführungsphase ist das resultierende Segment das Stapelsegment.   Sofern keine Kombinationsmethode angegeben ist, wird jedes Segment standardmäßig als PUBLIC betrachtet.  Das Argument *Verwendung* (*use*) ermöglicht die Auswahl einer anderen Wortgröße als die 16-Bit-Dimension, was impliziert ist, wenn dieses Argument nicht angegeben wird.  Das Argument *'class'* soll die Reihenfolge zulassen, in der der Linkeditor die Segmente im Speicher ablegt. Alle Segmente mit derselben Klasse werden unabhängig von ihrer Reihenfolge im Quellcode in einem angrenzenden Speicherblock abgelegt. Es gibt keinen Standardinitialisierungswert für dieses Argument. Wenn nicht angegeben, ist es nicht definiert. Infolgedessen wird vermieden, alle so definierten Segmente in einem kontinuierlichen Block zu verketten. |
| segment code use32 class = CODE segment data use32 class = DATA | | |
| 3.2 Directiva ASSUME |  | 3.2 Die ASSUME-Direktive |
| Directiva **ASSUME** stabileşte care sunt segmentele active la un moment dat. Ea are sintaxa generală: |  | Die Direktive ASSUME bestimmt, welche Segmente zu einem bestimmten Zeitpunkt aktiv sind. Es hat die allgemeine Syntax: |
| **ASSUME** CS:*nume1*, SS:*nume2*, DS:*nume3*, ES:*nume4*  **ASSUME** CS:*Name1*, SS: *Name2*, DS: *Name3*, ES: *Name4* | | |
| unde ordinea specificărilor de după ASSUME nu este importantă, oricare şi oricâte dintre acestea putând să lipsească. Fiecare dintre *nume1*, *nume2*, *nume3* şi *nume4* este un nume de segment sau cuvântul rezervat NOTHING.  Rolul directivei ASSUME este de a preciza asamblorului regiştrii de segment care trebuie utilizaţi pentru calculul adreselor efective ale etichetelor si ale variabilelor folosite în program. Existenţa acestei directive este necesară, deoarece un program poate fi alcătuit din mai multe segmente şi este necesar ca asamblorul să cunoască în fiecare moment care sunt segmentele active.  Prefixarea explicită a etichetelor şi variabilelor cu numele registrului de segment corespunzător (deci utilizarea operatorului de specificare a segmentului) furnizează în mod imediat această informaţie, având prioritate în cazul respectiv faţă de asocierea declarată prin directiva ASSUME.  Prezenţa acestei directive nu este necesară (având doar caracter de documentare) în cazul în care programul nu accesează etichete şi variabile (astfel de programe sunt însă extrem de rare). De asemenea, dacă operanzii în cadrul accesărilor indirecte nu fac referiri la etichete (de exemplu [EBX + 2] sau [EBP + EDI]) atunci se foloseşte ca registru de segment SS dacă apare EBP şi respectiv EDS în celelalte cazuri. Aceste asocieri se fac automat, independent de ASSUME.  Este foarte important de reţinut faptul că rolul acestei directive nu este şi de a încărca registrele segment cu adresele corespunzătoare! |  | Wenn die Reihenfolge der Spezifikationen nach ASSUME nicht wichtig ist, fehlen möglicherweise alle Angaben. Jeder der *Name1*, *Name2*, *Name3* und *Name4* ist ein Segmentname oder das reservierte Wort NOTHING.  Die Aufgabe der Direktive ASSUME besteht darin, die Segmentregister zu der Assembler anzugeben, die für die Berechnung der tatsächlichen Adressen der im Programm verwendeten Bezeichnungen und Variablen verwendet werden soll. Das Vorhandensein dieser Direktive ist notwendig, da ein Programm aus mehreren Segmenten bestehen kann und die Assembly zu jedem Zeitpunkt wissen muss, welche Segmente aktiv sind.  Das explizite Präfixieren von Labels und Variablen mit dem entsprechenden Segmentregisternamen (also unter Verwendung des Segmentspezifikationsoperators) stellt diese Informationen sofort zur Verfügung und hat in diesem Fall Vorrang vor der von der ASSUME-Direktive deklarierten Zuordnung.  Das Vorhandensein dieser Direktive ist nicht erforderlich (nur zu Dokumentationszwecken), wenn das Programm nicht auf Labels und Variablen zugreift (solche Programme sind jedoch äußerst selten). Wenn die Operatoren in den indirekten Zugriffen nicht auf Bezeichnungen verweisen (z. B. [EBX + 2] oder [EBP + EDI]), wird sie auch als SS-Segmentregister verwendet, wenn in den anderen Fällen EBP beziehungsweise EDS erscheinen. Diese Verknüpfungen werden unabhängig von ASSUME automatisch hergestellt.  Es ist sehr wichtig zu bedenken, dass die Rolle dieser Direktive nicht ist auch Segmentregister mit den entsprechenden Adressen zu laden! |
| 3.3 Directive pentru definirea datelor |  | 3.3 Diektiven zur Datendefinition |
| *Definirea* datelorînseamnă specificarea atributelor acestora şi alocarea spaţiului de memorie necesar.Deci definirea datelor este operaţia de *declarare* (specificarea atributelor) împreună cu cea de *alocare* (rezervarea a spaţiului de memorie necesar).  În cadrul declarării datelor, principalul atribut specificat este *tipul datei* = *dimensiunea de reprezentare –* octet, cuvânt, dublucuvânt, quadword). Alocarea datelor se face în cadrul segmentului în care acestea au fost declarate.  Forma generală a unei linii sursă în cazul unei declaraţii de date este: |  | Das *Definieren* der Daten bedeutet, ihre Attribute anzugeben und den erforderlichen Speicherplatz zuzuweisen. Das Definieren der Daten erfolgt also durch die *Deklarationsoperation* (Angabe der Attribute) zusammen mit der *Zuordnungsoperation* (Reservierung des erforderlichen Speicherplatzes).  In der Datendeklaration wird als Hauptattribut *der Typ des Datums* angegeben = *die Dimension der Darstellung* (Byte, Wort, Doppelwort, Vierfachwort). Die Datenzuordnung erfolgt innerhalb des Segments, in dem sie deklariert wurden.  Die allgemeine Form einer Quellzeile bei einer Datendeklaration ist: |
| [*Name*] *Date\_Typ Liste\_der\_Ausdrücke* [;*Kommentar*]  [nume] tip\_date lista\_expresii [;comentariu]  oder  [*Name*] *Typ\_Zuteilung Faktor* [;*Kommentar*]  [nume] tip\_alocare factor [;comentariu]  oder  [*Name*] **TIMES** *Faktor Date\_Typ Liste\_der\_Ausdrücke* [;*Kommentar*] | | |
| unde *nume* este o etichetă prin care va fi referită data. Tipul rezultă din tipul datei (dimensiunea de reprezentare) iar valoarea este adresa la care se va găsi în memorie primul octet rezervat pentru data etichetată cu numele respectiv.  *factor* este un număr care indică de câte ori se repetă lista de expresii care urmează în paranteză.  *Tip\_data* este o directivă de definire a datelor, una din următoarele:  **DB** - date de tip octet (BYTE)  **DW** - date de tip cuvânt (WORD)  **DD** - date de tip dublucuvânt (DWORD)  **DQ** - date de tip 8 octeţi (QWORD - 64 biţi)  **DT** - date de tip 10 octeţi (TWORD - utilizate pentru memorarea constantelor BCD sau constantelor reale de precizie extinsă).  De exemplu, secvenţa următoare defineşte şi iniţializează cinci variabile de memorie:  Data segment  var1 DB 'd' ;1 octet  .a DW 101b ;2 octeţi  var2 DD 2bfh ;4 octeţi  .a DQ 307o ;8 octeţi (1 quadword)  .b DT 100 ;10 octeţi  Variabilele var1 şi var2 sunt definite folosind etichete obişnuite, cu vizibilitate la nivelul întregului cod sursă, în timp ce .a şi .b sunt etichete locale, accesul la aceste variabile impunând următoarele constrângeri:   * acestea se pot accesa cu numele local, adică .a sau .b, până în momentul definirii unei alte etichete obişnuite (ele fiind locale etichetei ce le preced); * pot fi accesate de oriunde prin numele lor complet: var1.a, var2.a sau var2.b.   Valoarea de iniţializare poate fi şi o expresie, ca de exemplu în  vartest DW (1002/4+1)  După o directivă de definire a datelor pot să apară mai multe valori, permiţându-se astfel declararea şi iniţializarea de tablouri. De exemplu, declaraţia  Tablou DW 1,2,3,4,5  creează un tablou de 5 întregi reprezentaţi pe cuvinte având valorile respectiv 1, 2, 3, 4, 5. Dacă valorile de după directivă nu încap pe o singură linie se pot adăuga oricâte linii este necesar, linii ce vor conţine numai directiva şi valorile dorite. Exemplu:  Tabpatrate DD 0, 1, 4, 9, 16, 25, 36  DD 49, 64, 81  DD 100, 121, 144, 169  *Tip\_alocare* este o directivă de rezervare de date neiniţializate:  **RESB** – date de tip octet (BYTE)  **RESW** – date de tip cuvânt (WORD)  **RESD** – date de tip dublucuvânt (DWORD)  **RESQ** – date de tip 8 octeţi (QWORD – 64 biţi)  **REST** – date de tip 10 octeţi (TWORD – 80 biţi)  *factor* este un număr care indică de câte ori se repetă tipul alocării precizate.  De exemplu:  Tabzero RESW 100h  rezervă 256 de cuvinte pentru tabloul *Tabzero*.  NASM nu suportă sintaxa din MASM/TASM pentru rezervarea de spaţiu neiniţializat scriind DW ? sau ceva similar. În NASM se procedează altfel: operandul unei pseudo-instrucţiuni de tip RESB este o **expresie critică** **(**toţi operanzii care intervin în calcul trebuie să fie cunoscuţi în momentul în care expresia este întâlnită).  Exemplu:  buffer: **resb** 64; rezervă 64 octeţi  wordvar: **resw** 1 ; rezervă un cuvânt  realarray **resq** 10; vector de 10 numere reale  **Directiva TIMES**  permite asamblarea repetată a unei instrucţiuni sau definiţii de dată: |  | Wobei *Name* ein Label ist, die sich auf das Datum bezieht. Der Typ ergibt sich aus dem Datumstyp (Darstellungsgröße) und der Wert ist die Adresse, an der das erste Byte, das für das mit diesem Namen gekennzeichnete Datum reserviert ist, im Speicher abgelegt wird.  *Faktor* ist eine Zahl, die angibt, wie oft die Liste der Ausdrücke in Klammern wiederholt wird.  *Date\_Typ* ist eine Datendefinitionsanweisung, eine der folgenden:  **DB** – Bytedaten (BYTE)  **DW** – Wortdaten (WORD)  **DD** – doppelwort Datentyp (DWORD)  **DQ** – Datentyp 8 Bytes (QWORD – 64 Bits)  **DT** – Datentyp 10 Bytes (TWORD – zum Speichern von BCD-Konstanten oder reellen Konstanten mit erweiterter Genauigkeit).  Die folgende Sequenz definiert und initialisiert beispielsweise fünf Speichervariablen:  Data segment  var1 DB 'd' ;1 Byte  .a DW 101b ;2 Bytes  var2 DD 2bfh ;4 Bytes  .a DQ 307o ;8 Bytes (1 quadword)  .b DT 100 ;10 Bytes  Die Variablen var1 und var2 werden mit gemeinsamen Labels definiert, die im gesamten Quellcode sichtbar sind, während .a und .b lokale Labels sind. Der Zugriff auf diese Variablen unterliegt den folgenden Einschränkungen:   * Auf sie kann mit dem lokalen Namen, dh .a oder .b, bis zum Zeitpunkt der Definition eines anderen gemeinsamen Labels zugegriffen werden (sie sind lokal für das vorhergehende Label). * können von überall mit ihrem vollständigen Namen abgerufen werden: var1.a, var2.a oder var2.b.   Der Initialisierungswert kann auch ein Ausdruck sein, z. B. in  vartest DW (1002/4 + 1)  Nach einer Direktive, die die Daten definiert, können mehrere Werte erscheinen, wodurch die Deklaration und Initialisierung von Tabellen ermöglicht wird. Zum Beispiel die Aussage  Tablou DW 1,2,3,4,5  erstellt eine Tabelle mit 5 Ganzzahlen, die durch Wörter mit den Werten 1, 2, 3, 4, 5 dargestellt werden. Wenn die Werte nach der Direktive nicht in eine einzelne Zeile passen, können Sie so viele Zeilen wie nötig hinzufügen, die nur die Direktive und die gewünschten Werte enthalten. Beispiel:  Tabpatrate DD 0, 1, 4, 9, 16, 25, 36  DD 49, 64, 81  DD 100, 121, 144, 169  *Typ\_Zuteilung* ist eine nicht initialisierte Datenreservierungsanweisung:  **RESB** – Bytedaten (BYTE)  **RESW** – Worttypdaten (WORD)  **RESD** – doppelter Datentyp (DWORD)  **RESQ** – Datentyp 8 Bytes (QWORD - 64 Bits)  **REST** - Datentyp 10 Bytes (TWORD - 80 Bits)  Der *Faktor* ist eine Zahl, die angibt, wie oft der angegebene Zuordnungstyp wiederholt wird.  Zum Beispiel:  Tabzero RESW 100h  256 Wörter für das Tabzero-Gemälde.  Unterstützt NASM die MASM / TASM-Syntax zum Buchen von nicht initialisiertem Speicherplatz durch Schreiben von DW nicht? oder so ähnlich. In NASM ist die Vorgehensweise anders: Der Operand eines RESB-Pseudobefehls ist ein **kritischer Ausdruck** (alle in die Berechnung eingreifenden Operanden müssen zum Zeitpunkt des Auftretens des Ausdrucks bekannt sein).  Beispiel:  Buffer: **resb** 64; reserviert 64 Bytes  wordvar: **resw** 1; reserviere ein Wort  realarray **resq** 10; Vektor von 10 reellen Zahlen  **Die TIMES-Direktive** ermöglicht das wiederholte Zusammenstellen einer Anweisung oder von Datumsdefinitionen: |
| **TIMES** *Faktor Date\_Typ Ausdruck* (expresie) | | |
| De exemplu: |  | Zum Beispiel: |
| Tabchar **TIMES** 80 DB ‘a’ | | |
| creează un tablou de 80 de octeţi iniţializaţi fiecare cu codul ASCII al caracterului 'a'. |  | erstellt ein Array von 80 Bytes, die jeweils mit dem ASCII-Code des Zeichens 'a' initialisiert werden. |
| matrice10x10 **times** 10\*10 dd 0 | | |
| va furniza 100 de dublucuvinte dispuse continuu în memorie începând de la adresa asociată etichetei *matrice10x10*.  TIMES poate fi aplicată şi instrucţiunilor: |  | liefert 100 Doppelworte, die fortlaufend im Speicher angeordnet sind, beginnend mit der Adresse, die dem Label *matrice10x10* zugeordnet ist.  Die TIMES-Direktive kann auch auf die Anweisungen angewendet werden: |
| **TIMES** *Faktor Anweisung*  **TIMES** 32 **add** EAX, EDX; | | |
| ceea ce are ca efect EAX = EAX + 32\*EDX. |  | das hat die wirkung EAX = EAX + 32\*EDX. |
| 3.4 Directiva EQU |  | 3.4 Die EQU-Direktive |
| *Directiva* **EQU** permite atribuirea, în faza de asamblare, unei valori numerice sau şir de caractere unei etichete fără alocarea de spaţiu de memorie sau generare de octeţi. Sintaxa directivei EQU este |  | Die *EQU-Direktive* ermöglicht die Zuweisung eines numerischen Werts oder einer Zeichenfolge zu einem Label in der Assembler-Phase, ohne Speicherplatz zuzuweisen oder Bytes zu generieren. Die Syntax der EQU-Direktive lautet |
| *Name*  **EQU** *Ausdruck* | | |
| Exemple:  END\_OF\_DATA EQU '!'  BUFFER\_SIZE EQU 1000h  INDEX\_START EQU (1000/4 + 2)  VAR\_CICLARE EQU i  Prin utilizarea de astfel de echivalări textul sursă poate deveni mai lizibil. Se observă asemănarea etichetelor echivalate prin directiva EQU cu constantele din limbajele de programare de nivel înalt.  Expresia pentru echivalarea unei etichete definite prin directiva EQU poate conţine la rândul ei etichete definite prin EQU: |  | Beispiele:  END\_OF\_DATA EQU '!'  BUFFER\_SIZE EQU 1000h  INDEX\_START EQU (1000/4 + 2)  VAR\_CICLARE EQU i  Durch die Verwendung solcher Entsprechungen kann der Quelltext besser lesbar werden. Wir beobachten die Ähnlichkeit der durch die EQU-Direktive äquivalenten Bezeichnungen mit den Konstanten der höheren Programmiersprachen.  Der Ausdruck für die Gleichwertigkeit eines in der EQU-Richtlinie definierten Label kann auch in der EQU definierte Labels enthalten: |
| TABLE\_OFFSET EQU 1000h  INDEX\_START EQU (TABLE\_OFFSET + 2)  DICTIONAR\_STAR EQU (TABLE\_OFFSET + 100h) | | |
| 3.5 Directivele LABEL şi PROC |  | 3.5 LABEL- und PROC-Direktiven |
| Directiva **LABEL** permite numirea unei locaţii fără alocarea de spaţiu de memorie sau generare de octeţi, precum şi accesul la o dată utilizând alt tip decât cel cu care a fost definită data respectivă. Sintaxa este: |  | Die **LABEL-Direktive** ermöglicht die Benennung eines Speicherorts ohne Zuweisung von Speicherplatz oder Byte-Generierung sowie den Zugriff auf ein Datum mit einem anderen Typ als dem, mit dem das Datum definiert wurde. Die Syntax lautet: |
| *Name* **LABEL** *Typ* | | |
| unde *nume* este un simbol care nu a fost definit anterior în textul sursă, iar *tip* descrie dimensiunea de interpretare a simbolului şi dacă acesta se va referi la cod sau la date.  Numele primeşte ca valoare contorul de locaţii. Tip poate să fie una din următoarele: |  | Dabei ist *Name* ein Symbol, das zuvor nicht im Quelltext definiert wurde, und der Typ beschreibt die Interpretationsdimension des Symbols und ob es sich auf den Code oder die Daten bezieht.  Der Name erhält den Wert des Programmzählers. Der Typ kann einer der folgenden sein: |
| |  |  |  | | --- | --- | --- | | **BYTE** | **NEAR** | **FAR** | | **WORD** | **QWORD** | **PROC** | | **DWORD** | **TBYTE** | **UNKNOWN** | | | |
| Tipurile **BYTE**, **WORD**, **DWORD**, **QWORD** şi **TBYTE** etichetează respectiv date de 1, 2, 4, 8 şi 10 octeţi. Iată un exemplu de iniţializare a unei variabile de memorie ca pereche de octeţi însă accesată ca şi cuvânt: |  | Die Typen **BYTE**, **WORD**, **DWORD**, **QWORD** und **TBYTE** kennzeichnen jeweils Daten von 1, 2, 4, 8 und 10 Bytes. Hier ist ein Beispiel für die Initialisierung einer Speichervariablen als Bytepaar, auf das jedoch als Wort zugegriffen wird: |
| |  |  | | --- | --- | | data segment  …  Varcuv LABEL WORD  DB 1,2  …  data ends | code segment  …  **mov** AX, Varcuv  … | | | |
| Tipul UNKNOWN declară un tip necunoscut şi este folosit atunci când se doreşte să existe posibilitatea accesării unei variabile de memorie în mai multe moduri (se aseamănă cu tipul *void* din limbajul C). De exemplu, accesarea variabilei *tempvar* din secvenţa de mai jos uneori ca octet şi alteori ca şi cuvânt poate fi realizată prin declararea ei ca etichetă de tip UNKNOWN: |  | Der UNKNOWN-Typ deklariert einen unbekannten Typ und wird verwendet, wenn auf eine Speichervariable auf verschiedene Arten zugegriffen werden soll (ähnelt dem *void*-Typ in der C-Sprache). Der Zugriff auf die Variable *tempvar* in der folgenden Reihenfolge kann beispielsweise manchmal als Byte und manchmal als Wort erfolgen, indem Sie sie als UNKNOWN-Tag deklarieren: |
| |  |  | | --- | --- | | data segment  …  tempvar LABEL UNKNOWN  DB ?,?  …  data ends | code segment  …  **mov** tempvar, AX; verwenden als Wort  …  **add** DL, tempvar; verwenden als Byte  … | | | |
| O altă soluţie pentru adresarea unei date cu un alt tip decât cel cu care a fost declarată este utilizarea operatorului de conversie **PTR**.  Noţiunea de subrutină oferă posibilitatea dezvoltării modulare a programelor, permiţând concentrarea atenţiei asupra punctelor esenţiale ale unui program, ignorându-se în acele locuri detaliile de implementare. De asemenea, prezenţa subrutinelor aduce avantajul reutilizării codului (deci obţinerea în final a unui cod cât mai compact), o subrutină fiind scrisă o singură dată, putând fi însă apelată de oricâte ori şi în orice punct al programului cu valori diferite ale eventualilor parametri.  În limbajul de asamblare 8086 definirea unei subrutine începe cu o directivă **PROC**: |  | Eine andere Lösung für die Adressierung von Daten eines anderen Typs als dem, mit dem sie deklariert wurden, ist die Verwendung des **PTR**-Konvertierungsoperators.  Der Begriff der Subroutine bietet die Möglichkeit der modularen Entwicklung von Programmen, die es ermöglicht, sich auf die wesentlichen Punkte eines Programms zu konzentrieren und die Einzelheiten der Implementierung an diesen Stellen zu ignorieren. Das Vorhandensein von Unterprogrammen bringt auch den Vorteil mit sich, dass der Code wiederverwendet wird (um einen möglichst kompakten Code zu erhalten), wobei ein Unterprogramm nur einmal geschrieben wird, aber mit unterschiedlichen Werten der möglichen Parameter jederzeit und an jedem Punkt des Programms aufgerufen werden kann.  In der Assemblersprache 8086 beginnt die Definition eines Unterprogramms mit einer **PROC-Anweisung**: |
| <*Prozedurname*> **PROC** [*Aufruftyp*] | | |
| unde *nume\_procedură* este o etichetă reprezentând numele procedurii, iar *tip\_apel* este NEAR sau FAR. Dacă lipseşte, atunci se consideră implicit NEAR. Procedura va fi NEAR dacă va fi apelată numai în cadrul segmentului de cod în care este definită. Procedura va fi FAR dacă va fi apelată şi din alte segmente de cod.  Directiva **ENDP** marchează sfârşitul unei subrutine ce începe cu PROC. Sintaxa ei este: |  | Dabei ist *Prozedurname* ein Label, das den Prozedurnamen darstellt, und der *Aufruftyp* ist NEAR oder FAR. Wenn es fehlt, wird es standardmäßig als NEAR betrachtet. Die Prozedur ist NEAR, wenn sie nur innerhalb des Codesegments aufgerufen wird, in dem sie definiert ist. Die Prozedur ist FAR, wenn sie aus anderen Codesegmenten aufgerufen wird.  Die **ENDP-Direktive** markiert das Ende eines Unterprogramms, das mit PROC beginnt. Ihre Syntax lautet: |
| <*Prozedurname*> **ENDP** | | |
| între cele două directive se vor scrie instrucţiunile şi directivele corpului procedurii. |  | Zwischen den beiden Direktiven werden die Anweisungen und Direktiven des Hauptteils des Prozedur geschrieben. |
| 3.6 Blocuri repetitive |  | 3.6 Wiederholungsblöcke |
| Un bloc repetitiv este o construcţie prin care i se cere asamblorului să genereze în mod repetat o configuraţie de octeţi. Principalele blocuri repetitive sunt **REPT**, **IRP** şi **IRPC**.  Un bloc repetitiv delimitat de directivele **REPT** şi **ENDM** are următoarea sintaxă de definire: |  | Ein repetitiver Block ist ein Konstrukt, bei dem das Assembler wiederholt eine Bytekonfiguration generieren muss. Die Hauptwiederholungsblöcke sind **REPT**, **IRP** und **IRPC**.  Ein sich wiederholender Block, der durch die Anweisungen **REPT** und **ENDM** begrenzt ist, hat die folgende Definitionssyntax: |
| **REPT** *Zähler* (contor)  *Sequenz* (secvenţă)  **ENDM** | | |
| cu semnificaţia că secvenţa va fi asamblată de *contor* ori. De exemplu, secvenţele: |  | Dies bedeutet, dass die Sequenz *Zähler* malen assembliert wird. Zum Beispiel die Sequenzen: |
| |  |  |  | | --- | --- | --- | | **Rept** 5  Dw 0  **endm** | und | dw 0  dw 0  dw 0  dw 0  dw 0 | | | |
| generează acelaşi cod, lucru ce se poate realiza şi cu: dw 5 DUP (0)  Exemplul următor însă nu are un echivalent atât de simplu. El generează 5 locaţii de memorie consecutive conţinând valorile de la 0 la 4. Folosim în acest scop şi directiva =: |  | erzeugt den gleichen Code, der gemacht werden kann mit: dw 5 DUP (0)  Das folgende Beispiel hat jedoch kein so einfaches Äquivalent. Es erzeugt 5 aufeinanderfolgende Speicherplätze mit Werten von 0 bis 4. Wir verwenden auch die Direktive =: |
| |  |  |  | | --- | --- | --- | | Intval = 0  **Rept** 10  Dw Intval  Intval = Intval + 1  **endm** | generieren werden | dw 0  dw 1  dw 2  dw 3  dw 4 | | | |
| Blocurile repetitive pot fi imbricate. Secvenţa : |  | Wiederholte Blöcke können verschachtelt werden. Die Sequenz: |
| **REPT** 5  **REPT** 2  Sequenz  **ENDM**  **ENDM** | | |
| generează de 10 ori secvenţa specificată.  Directiva **IRP** are sintaxa: |  | Erzeugt 10 malen der angegebenen Sequenz.  Die **IRP**-Direktive hat die Syntax: |
| **IRP**  *Parameter*, <*arg1* [*arg2*]…>  *Sequenz*  **ENDM** | | |
| Se efectuează în mod repetat asamblarea secvenţei, câte o dată pentru fiecare argument prevăzut în lista de argumente, prin înlocuirea textuală în secvenţă a fiecărei apariţii a parametrului cu argumentul curent. Argumentele pot fi şiruri de caractere, simboluri, valori numerice. De exemplu: |  | Die Assemblierung der Sequenz wird einmal für jedes in der Argumentliste angegebene Argument wiederholt, indem jedes Vorkommen des Parameters durch das aktuelle Argument ersetzt wird. Argumente können Zeichenfolgen, Symbole und numerische Werte sein. Zum Beispiel: |
| |  |  |  | | --- | --- | --- | | **IRP**  param, <0, 1, 4, 9, 16, 25>  db param  **ENDM** | generieren werden die Sequenz | db 0  db 1  db 4  db 9  db 16  db 25 | | | |
| Iar |  | und |
| |  |  |  | | --- | --- | --- | | **IRP**  reg,<AX,BX,CX,DX>  **mov** reg, DI  **ENDM** | generieren werden die Sequenz | **mov** AX,DI  **mov** BX,DI  **mov** CX,DI  **mov** DX,DI | | | |
| Directiva **IRPC** are un efect similar, ea realizând însă înlocuirea textuală a parametrului, pe rând, cu fiecare caracter dintr-un şir de caractere dat. Sintaxa ei este: |  | Die **IRPC**-Direktive hat eine ähnliche Wirkung, führt jedoch die textuelle Ersetzung des Parameters nacheinander durch jedes Zeichen in einer bestimmten Zeichenfolge durch. Ihre Syntax lautet: |
| **IRPC**  *Parameter*, *Zeichenfolge*  *Sequenz*  **ENDM** | | |
| De exemplu: |  | Zu Beispiel |
| **IRPC** nr,1375  db nr  **ENDM** | | |
| creează 4 octeţi având respectiv valorile 1, 3, 7 şi 5. |  | Erstellt 4 Bytes mit den Werten 1, 3, 7 und 5. |
| 3.7 Directiva INCLUDE |  | 3.7 Die INCLUDE-Direktive |
| Directiva **INCLUDE** are sintaxa |  | Die INCLUDE-Direktive hat die folgende Syntax: |
| **INCLUDE** *Name\_Datei* | | |
| Efectul ei (similar cu cel al directivei #*include* a preprocesorului C) este de a insera textual fişierul *nume\_fişier* în textul sursă curent. Inserarea se face în locul în care apare directiva INCLUDE respectivă. *nume\_fişier* este specificarea unui nume de fişier DOS, putând aşadar să conţină specificări de unitate de disc, directoare, nume de fişier şi tip. În lipsa specificării unui tip se consideră implicit .ASM. Exemplu: |  | Seine Wirkung (ähnlich der der #*include*-Direktive von der Präprozessor C) besteht darin, den *Name\_Datei* Datei textuell in den aktuellen Quelltext einzufügen. Das Einfügen erfolgt an der Stelle, an der die INCLUDE-Direktive erscheint. *Name\_Datei* ist die Angabe eines DOS-Dateinamens, der Angaben zu Laufwerk, Verzeichnis, Dateiname und Typ enthalten kann. Wenn Sie keinen Typ angeben, wird standardmäßig .ASM berücksichtigt. Beispiel: |
| **cod** segment  **mov** ax,1  **INCLUDE** INSTR2.ASM  **push** AX | | |
| iar fişierul *instr2.asm* conţine codul |  | und die instr2.asm-Datei enthält den Code |
| **mov** BX,3  **add** AX,BX  **dec** BX | | |
| Rezultatul asamblării fişierului prog.asm va fi echivalent cu cel al asamblării codului |  | Das Ergebnis der Assemblierung der Datei prog.asm entspricht dem Assemblierung von der Code |
| **cod** segment  **mov** ax,1  **mov** BX,3  **add** AX,BX  **dec** BX  **push** AX | | |
| Fişierele incluse pot conţine la rândul lor alte directive INCLUDE, ş.a.m.d. până la orice nivel, instrucţiunile respective fiind inserate corespunzător pentru crearea în final a unui singur cod sursă. |  | Die enthaltenen Dateien können auch andere Direktiven wie INCLUDE usw. enthalten bis zu einer beliebigen Ebene, wobei die entsprechenden Anweisungen ordnungsgemäß für die endgültige Erstellung eines einzelnen Quellcodes eingefügt werden. |
| 3.8 Macrouri |  | 3.8 Makros |
| Un *macro* este un text parametrizat căruia i se atribuie un nume. Ori de câte ori găseşte numele, asamblorul pune în codul sursă textul cu parametrii actualizaţi. Operaţia este cunoscută şi sub numele de *expandarea macroului*.  Se poate face o analogie cu directiva INCLUDE prezentată anterior. Faţă de fişierele incluse, macrourile prezintă un grad sporit de flexibilitate permiţând transmiterea de parametri şi existenţa de etichete locale.  Un macro este delimitat de directivele **MACRO** şi **ENDM** conform următoarei sintaxe: |  | Ein Makro ist ein parametrisierter Text, dem ein Name zugewiesen wird. Immer wenn der Name gefunden wird, fügt der Assembler den Text mit den aktualisierten Parametern in den Quellcode ein. Die Operation wird auch als *Makroerweiterung* bezeichnet.  Eine Analogie kann mit der oben dargestellten INCLUDE-Direktive hergestellt werden. Makros sind im Vergleich zu den enthaltenen Dateien flexibler und ermöglichen die Übertragung von Parametern und das Vorhandensein lokaler Labels.  Ein Makro wird durch die Direktiven **MACRO** und **ENDM** gemäß der folgenden Syntax begrenzt: |
| *Name* **MACRO**  [*Parameter* [.*Parameter*]...]  *Körper Anweisungen*  **ENDM** | | |
| De exemplu, pentru interschimbarea valorilor a două variabile cuvânt putem defini următorul macro: |  | Für den Werteaustausch zweier Wortvariablen können wir beispielsweise folgendes Makro definieren: |
| swap **MACRO** a,b  **mov** ax, a  **mov** a,b  **mov** b,ax  **ENDM** | | |
| Pentru înmulţirea cu 4 a valorii unei variabile (rezultatul depunându-se în DX:AX) se poate scrie următorul macro: |  | Zum Multiplizieren des Werts einer Variablen mit 4 (das Ergebnis wird in DX: AX abgelegt) kann das folgende Makro geschrieben werden: |
| inmcu4 **MACRO** a  **mov** AX, a  **sub** DX, DX  **shl** AX,1  **rcl** DX, 1  **shl** AX,1  **rcl** DX, 1  **ENDM** | | |
| Macrourile pot conţine blocuri repetitive.  O posibilă problemă ce apare se referă la definirea unei etichete într-un macro. Să presupunem că se defineşte o etichetă şi că în program există mai mult de un apel al acelui macro. Eticheta definită va apărea la fiecare expandare a macroului în codul programului, cauzând o eroare de „redefinire de etichetă”.  Soluţia unei astfel de probleme este oferită de către directiva **LOCAL**, care, la apariţia ei în cadrul unui macro, forţează ca domeniul de vizibilitate al etichetelor specificate ca argumente să fie numai acel macro.  Utilizarea directivei LOCAL trebuie să urmeze imediat directivei MACRO. Numărul argumentelor nu este limitat.  În cadrul utilizării macrourilor, referinţele anticipate (*forward references*) nu sunt permise. Macrourile trebuie să fie definite întotdeauna înaintea invocării. De asemenea, macrourile pot fi imbricate.  Macrourile pot conţine blocuri repetitive. De asemenea, macrourile pot invoca la rândul lor alte macrouri. |  | Makros können sich wiederholende Blöcke enthalten.  Ein mögliches Problem ist das Definieren einem Label in einem Makro. Angenommen, ein Label ist definiert und es gibt mehr als einen Aufruf in diesem Makro im Programm. Das definierte Tag wird bei jeder Makroerweiterung im Programmcode angezeigt, was zu einem Fehler bei der Neudefinition dem Label führt.  Die Lösung eines solchen Problems bietet die **LOCAL**-Direktive, die in einem Makro das Sichtbarkeitsfeld der als Argumente angegebenen Labels auf dieses Makro beschränkt.  Die Verwendung der LOCAL-Direktive muss unmittelbar nach der MACRO-Direktive erfolgen. Die Anzahl der Argumente ist nicht begrenzt.  Bei der Verwendung von Makros sind Vorwärtsreferenz (*forward references*) nicht zulässig. Makros müssen immer vor dem Aufruf definiert werden. Makros können auch verschachtelt werden.  Makros können sich wiederholende Blöcke enthalten. Makros können auch andere Makros aufrufen. |