Dyrektywy asemblera

W asemblerze programy pisze się zazwyczaj według pewnego szablonu, deklaruje się segmenty kodu, stosu i danych. Operacje na elementach danych są wykonywane przy użyciu instrukcji procesora lecz gdyby nie dyrektywy asemblera pisanie programów byłoby niezwykle uciążliwe. Ułatwiają one programiście całe zadanie umożliwiając wybranie grup instrukcji procesora, modelu pamięci, definicję etykiet, etc.. W celu deklaracji tych elementów używa się specjalnych dyrektyw:

Dyrektywy docelowego procesora - pozwalają deklarować grupy instrukcji procesora, które będą używane przez program. Dyrektywa docelowego procesora powinna być podana na początku programu, tak aby objąć całą jego zawartość. Domyślnym rodzajem procesora jest 8086 lecz grupa instrukcji obsługiwanych przez ten procesor jest dość skromna. Można więc zadeklarować inny typ procesora, dostępne są następujące opcje:

- .8086 - asemblowanie wszystkich nieuprzywilejowanych rozkazów 8086, 8088, 80186 oraz rozkazów koprocesora 8087
- .186 - asemblowanie wszystkich nieuprzywilejowanych poleceń procesora 80186
- .287 - włącza asemblowanie rozkazów koprocesora 80287
- .386 - umożliwia asemblowanie wszystkich nieuprzywilejowanych rozkazów procesora 80386
- .387 - włącza asemblowanie rozkazów koprocesora 80387
- .486 - umożliwia asemblowanie wszystkich nieuprzywilejowanych rozkazów procesora 80486
- .586 - umożliwia asemblowanie nieuprzywilejowanych rozkazów procesorów Pentium

Aby móc asemblować programy zawierające rozkazy uprzywilejowane, pracujące w trybie chronionym, należy użyć innych dyrektyw docelowego procesora:

.286p
.386p
.486p
.586p

.MODEL - powoduje wybranie standardowego modelu pamięci używanego przez program. Model pamięci to konfiguracja decydująca o sposobie użycia i łączenia segmentów. Każdy model cechuje się innymi ograniczeniami w zakresie maksymalnej przestrzeni dostępnej dla kodu i danych. Dobór modelu pamięci ma decydujący wpływ na sposób dostępu do zmiennych oraz procedur. Asembler obsługuje następujące modele pamięci:

Tiny - Łączna wielkość kodu i danych nie może przekroczyć 64kB
Small - Segment kodu max. 64kB, segment danych max. 64kB, jeden segment kodu i jeden danych
Medium - Segment danych max. 64kB, kod dowolnej wielkości, wiele segmentów kodu jeden segment danych
Compact - Segment kodu max. 64kB, dane dowolnej wielkości, wiele segmentów danych jeden segment kodu
Large - Segment kodu dowolnej wielkości, segment danych dowolnej wielkości, wiele segmentów kodu i danych
Huge - Podobnie jak w modelu Large ale pojedyncze zmienne mogą być większe niż 64kB
Flat - Brak segmentów, 32-bitowe adresy dla kodu i danych, tylko tryb chroniony

ASSUME - przypisuje etykietę do wskazanego rejestru segmentowego. Dyrektywa może być używana w dowolnym miejscu w programie, np.:

```
ASSUME cs: CODE
```

Przypisujemy w ten sposób segment kodu do rejestru CS. Odtąd gdy będziemy się odwoływać się do etykiety CODE, będziemy odwoływać się do rejestru kodu - CS.

Aby unieważnić to powiązane należy użyć słowa kluczowego *NOTHING*:

```
ASSUME cs: NOTHING
```

Aby unieważnić wszystkie powiązania:

```
ASSUME NOTHING
```

Dyrektywa ASSUME nie zmienia zawartości rejestrów segmentowych dlatego programista musi ustawić odpowiedni rejestr segmentowy samodzielnie w dalszej części programu.

SEGMENT, ENDS - dyrektywy służące do deklaracji segmentów. Używa się ich w następujący sposób:

```
nazwa_segmentu SEGMENT [typ_segmentu] [połączenie]
[wielkosc_slowa] ['klasa']
```

gdzie:

nazwa_segmentu - jest to dowolna nazwa przyjmowana przez kompilator. Może być później w programie wykorzystywana jako identyfikator segmentu.

typ_segmentu - określa sposób przydzielania pamięci:

Byte - adres dowolny - segment ładowany jest w dowolnym miejscu

Word - adres parzysty - segment ładowany jest na granicy pełnego słowa

Para - adres podzielny przez 16 - segment ładowany jest na granicy pełnego paragrafu

Page - adres podzielny przez 256 - segment ładowany jest na granicy strony (1 strona=256 bajtów)

połączenie - określa jak kompilator ma łączyć segmenty o tej samej nazwie.

PRIVATE - domyślny typ łączenia, który oznacza iż segment nie zostanie połączony z żadnym innym.

PUBLIC - segmenty o tej samej nazwie są łączone w jeden ciągły segment. Wszystkie adresy w segmencie są łączone względem jego początku, np.:

```
DANE1 SEGMENT PUBLIC
ZMIENNA1 DB 0
```

```
ZMIENNA2 DB 0
DANE1 ENDS

DANE1 SEGMENT PUBLIC
ZMIENNA3 DB 0
ZMIENNA4 DB 0
DANE1 ENDS
```

zadeklarowanie tych dwóch segmentów jest równoważne:

```
DANE1 SEGMENT PUBLIC
ZMIENNA1 DB 0
ZMIENNA2 DB 0
ZMIENNA3 DB 0
ZMIENNA4 DB 0
DANE1 ENDS
```

STACK - wszystkie segmenty o tej samej nazwie łączone są w jeden ciągły segment, który przy ładowaniu programu do pamięci inicjowany jest jako stos (SP pokazuje na ostatni bajt segmentu). Jeśli chce się definiować w programie segment stosu, trzeba zadeklarować dla niego parametr 'połączenie' - *STACK*. Jeśli zdefiniuje się stos bez parametru *STACK*, zajdzie potrzeba inicjowania rejestrów stosu w programie.

COMMON - nakłada wszystkie segmenty o tej samej nazwie, umieszczając początek każdego w tym samym miejscu. Powstaje w efekcie obszar o wielkości największego z segmentów, a zmienne mogą się nakładać.

MEMORY - umieszcza wszystkie segmenty o tej samej nazwie w najwyższym fizycznym segmencie pamięci. Jeśli jest więcej niż jeden segment *MEMORY* nakładane są one jak w przypadku *COMMON*.

AT adres - adresy wszystkich zmiennych i etykiet w segmencie są obliczane względem wartości segmentu podanej przy AT czyli utworzony zostaje segment pod adresem absolutnym.

klasa - zapewnia odpowiednią kolejność łączenia segmentów. Deklaracja typu klasy jest umieszczona znakami ' . Standardowe klasy to CODE, DATA i STACK. Segmenty o tym samym typie klasy są

ładowane wspólnie nawet w przypadku gdy wewnątrz programu nie są leżą obok siebie.

Przykładowy szablon programu w asemblerze może wyglądać następująco:

```
DSTACK SEGMENT STACK 'STACK'
;deklaracja segmentu stosu
DB 64 DUP ('STACK')
;wypełnienie stosu
DSTACK ENDS
;koniec segmentu

ASSUME CS:CODE, SS:DSTACK
;przypisanie rejestrów do etykiet (segmentów)
CODE SEGMENT
;deklaracja segmentu kodu
Start:
<kod programu>
CODE ENDS
;koniec segmentu z kodem

DATA SEGMENT
;deklaracja segmentu z danymi
<dane programu>
DATA ENDS
;koniec segmentu z danymi
END Start
```


ORG - Dyrektywa nadaje licznikowi adresów wartość wyrażenia. Wszystkie następujące po ORG adresy zaczynają się od zadeklarowanej wartości. Dyrektywa ta jest zazwyczaj używana do nadawania danym lub instrukcjom określonego przesunięcia w segmencie. Jeśli nasz program ma być zapisany w pliku COM, to musimy napisać ORG 100H - spowoduje to przesunięcie początku o 100H bajtów.

GROUP - Dyrektywa ta pozwala wiązać kilka segmentów zdefiniowanych dyrektywami SEGMENT tak, że adresy w każdym z nich

są liczone nie względem jego początku, ale względem początku grupy. Segmenty w grupie mogą być ciągłe (max. 64KB), ale nie muszą - między segmentami grupy mogą znajdować się segmenty nie należące do danej grupy. Deklaracja grupy wygląda następująco:

```
nazwa GROUP nazwa_segmentu
```

Aby ustalić adresację w obrębie całej grupy, należy użyć dyrektywy ASSUME, podając w niej nazwę grupy, np.:

```
Grupa1 GROUP Dane1, Dane2

Dane1 SEGMENT
Par1 DB ?
Dane1 ENDS

Dane2 SEGMENT
Par2 DW ?
Dane2 ENDS

KOD SEGMENT

ASSUME CS:KOD, DS:Grupa1
;segmenty Dane1 i Dane2 będą adresowane
;względem rejestru DS

MOV AX,Grupa1
;segment grupy Grupa1 do rejestru DS
MOV DS,AX
;...
SUB CX,Par2
;przesunięcie par liczone jest względem początku grupy
KOD ENDS
```

EVEN i **EVENDATA** - Dyrektywy te umieszczają instrukcję lub definicję danych na granicy pełnego słowa. EVEN wstawia NOP (90h) czyli instrukcję pustą i nie można jej używać w segmentach z parametrem '*typ_segmentu*' równym BYTE. EVENDATA działa analogicznie z tym że

nie wstawia instrukcji pustej lecz puste dane (00h). Dopasowanie kolejnego polecenia w segmencie kodu do parzystego przesunięcia 16-bitowego może zwiększyć wydajność aplikacji na procesorach z 16-bitową magistralą danych.

PUBLIC -Dyrektywa czyni podaną przy niej nazwę (etykietę, nazwę zmiennej lub procedury) dostępną dla innych modułów. Jest wymagana jeśli program zawiera dane które mają być udostępnione innym modułom. Składnia tej dyrektywy jest następująca:

```
PUBLIC nazwa

PUBLIC nazwa1, nazwa2, nazwa3, ...
```

EXTERN - Dyrektywa deklaruje nazwę użytą w module jako zewnętrzną, tj. taką, do której odwołanie zostanie rozwiązane na etapie łączenia modułów. Dyrektywa EXTERN ma następującą składnię:

```
EXTERN nazwa:typ

EXTERN nazwa1:typ, nazwa2:typ, nazwa3:typ, ...
```

Typ w tej deklaracji określa rodzaj symbolu i może być następujący: BYTE, WORD, DWORD, NEAR, FAR. Jeśli program został skompilowany z użyciem modelu pamięci TINY, SMALL lub COMPACT to typ symbolu zostaje domyślnie ustawiony na NEAR. W przypadku innych modeli typ jest ustawiany na FAR. Poszczególne dyrektywy EXTERN powinny znaleźć się we właściwych segmentach: deklaracje procedur w segmencie kodu, deklaracje danych w segmencie danych. Deklaracje stałych mogą znaleźć się w dowolnym miejscu w programie. Poniżej przedstawiony jest dwumodułowy przykład wykorzystujący dyrektywę EXTERN:

Pierwszy moduł wyglądać może następująco:

```
EXTERN PIERWSZA:FAR, ZMIENNA:BYTE

KOD1 SEGMENT

ASSUME CS:KOD1,DS:KOD1
```

```
START:
;...
CALL PIERWSZA
```
;wywołanie będzie typu FAR
```
...
MOV AL,ZMIENNA
```
;odwołanie do zmiennej z podaniem segmentu
```
KOD1 ENDS
END START
```

Drugi natomiast wygląda tak:

```
PUBLIC PIERWSZA,ZMIENNA
KOD2 SEGMENT

ASSUME CS:KOD2,DS:KOD2
;...
PIERWSZA PROC FAR
```
;procedura typu FAR
```
;...
RET
PIERWSZA ENDS
;...
ZMIENNA DB 0
;...
KOD2 ENDS
```

LABEL - Dyrektywa ta służy do tworzenia etykiet, przyporządkowując im aktualną wartość adresu. Jest ona użyteczna przy nadawaniu zmiennym pamięciowym, już nazwanym, innej nazwy z innym kodem. Etykiety deklarujemy według następującego schematu:

```
nazwa_etykiety LABEL typ
```

Wyróżnia się następujące typy etykiet:

BYTE
WORD
DWORD
QWORD

TBYTE
NEAR
FAR

Przykładowo:

```
SLOWO LABEL WORD
ZMIENNA1 DB 5h
ZMIENNA2 DB 8h
```

Liczby 5 i 8 znajdą się w dwóch kolejnych bajtach pamięci. Można odwoływać się do nich używając ich nazw (ZMIENNA1, ZMIENNA2) lub odwołać się do etykiety SLOWO, gdzie nadamy im wartość słowa (dwa bajty jednocześnie):

```
MOV AX,SLOWO          ;ładuje do AX wartość 0508H

MOV CL,ZMIENNA1       ;ładuje do CL wartość 5h
```

ENUM (tylko TASM) - pozwala na zdefiniowanie zestawu stałych automatycznie przydzielając im kolejne wartości całkowite. Składnia jest następująca:

```
nazwa ENUM symbol1, symbol2, symbol3, ...
```

Symbol1 otrzyma wartość 0, Symbol2 wartość 1, Symbol3 wartość2 itd.. Definicje symboli mogą być również umieszczone w kilku wierszach otoczonych nawiasami {}:

Procedury

Asembler, jak i inne języki programowania, daje nam możliwość tworzenia własnych procedur. Do tego celu używa się dyrektywy PROC i ENDP. Ogólna budowa deklaracji procedury jest następująca:

```
Nazwa PROC
...
kod procedury
...
RET
Nazwa ENDP
```

Instrukcja RET powoduje powrót do miejsca, z którego procedura została wywołana. Do wywołania procedury używa się instrukcji CALL.

```
procedura_1 PROC
...
ret
procedura_1 ENDP
START:
...
call procedura_1
...
call procedura_1
...
```

Parametrem CALL może być również adres przechowywany w rejestrze bazowym lub indeksowym (BX, BP, SI lub DI). Techniki tej używa się gdy zachodzi potrzeba zmiany adresu procedury w trakcie działania programu (wywołanie pośrednie). Adres procedury umieszczony w zmiennej (wskaźnik) należy skopiować do rejestru bazowego lub indeksowego, a następnie dokonać wywołania procedury używając instrukcji CALL.. Adresy wielu procedur wygodnie jest umieszczać w tablicy wskaźników:

```
TABLICA DW PROCEDURA_1, PROCEDURA_2, PROCEDURA_3

PROCEDURA_1 PROC
...
RET
PROCEDURA_1 ENDP

PROCEDURA_2 PROC
...
RET
PROCEDURA_2 ENDP

...

START:
...
MOV SI,BX
```

```
;w BX musi być liczba od 0 do 2
SHL SI,1
;przesunięcie adresu w tablicy
CALL NEAR PTR TABLICA[SI]
;wywołanie wybranej procedury
...
```

Jeżeli wywoływana procedura oraz instrukcja CALL znajdują się w tym samym segmencie kodu, stosowane jest wywołanie bliskie czyli typu NEAR. Jeśli natomiast polecenie CALL oraz procedura znajdują się w różnych segmentach kodu, stosowane jest wywołanie odległe typu FAR. Dyrektywa .MODEL automatycznie ustawia atrybut dla wszystkich procedur w programie. W przypadku modelu pamięci *Tiny*, *Small* i *Compact* stosowane są procedury NEAR, a w przypadku pozostałych typu FAR.

Aby procesor mógł powrócić z procedury, podczas wywoływania instrukcji CALL, na stosie zostaje położony rejestr IP (w przypadku procedury NEAR) lub CS oraz IP (dla procedur typu FAR). Po wykonaniu kodu procedury procesor powinien trafić na instrukcję RET (RETF dla procedury typu FAR), która ściąga ze stosu zapamiętany rejestr IP (CS i IP) i skacze do miejsca skąd procedura została wywołana (w następnej kolejności wykonywana jest pierwsza instrukcja po CALL).

Procedur nie można zagnieżdżać (definicja procedury nie powinna się pojawić wewnątrz innej procedury). Może natomiast wystąpić zagnieżdżone wywołanie procedur kiedy podprocedura wywołuje inne podprocedury.

Parametry procedury

W większości przypadków programy napisane w asemblerze przekazują argumenty do podprocedur poprzez rejestry. Metoda ta jest najwydajniejsza, ponieważ dostęp do rejestrów jest szybszy aniżeli do pamięci. Pamiętać należy tutaj jednak o tym, by zapisywać wartość używanych przez procedurę rejestrów na stosie przed jej wywołaniem, a po powrocie odtworzyć ich wartość (oczywiście z wyjątkiem rejestru, który zwraca wynik).

Jeśli jednak procedura jest bardzo rozbudowana i przyjmuje dużą ilość parametrów, niejednokrotnie programista jest zmuszony do przyjmowania parametrów procedury przez stos. Metoda ta jest często wykorzystywana przez języki wysokiego poziomu, ale może być z powodzeniem stosowana w programach napisanych w asemblerze. Turbo Assembler jest wyposażony w rozbudowaną dyrektywę PROC, która pozwala na definiowanie nazw i typów poszczególnych parametrów procedury:

```
nazwa_procedury PROC [jezyk]
ARG parametr1:typ, parametr2:typ, ... RETURNS
wynik1:typ, wynik2:typ, ...
```

Każdy parametr po dyrektywie ARG otrzymuje swój typ, np.: BYTE, WORD, DWORD itd.. Jeśli parametr ten zostanie pominięty, domyślnie wybierany jest typ WORD. Dostępne specyfikatory języka to: C, Pascal, Fortran lub Basic.

Poniżej przedstawiony jest przykład użycia rozbudowanej dyrektywy PROC:

```
suma PROC, Pascal
;definicja procedury dodającej 2 liczby całkowite
ARG zm1:WORD, zm2:word RETURNS wynik:WORD
;ustalenie parametrów procedury oraz wartości zwracanej przez procedurę
push ax
;zachwanie AX
mov ax,zm1
add ax,zm2
mov wynik,ax
;zmienna wynik będzie umieszczona na stosie
pop ax
ret
suma ENDP
```

Wywołanie tak napisanej procedury odbywa się za pomocą dyrektywy CALL (nazwa identyczna z instrukcją CALL):

```
.data
w1 dw 120h
w2 dw 46h
```

```
wynik_dodawania dw ?
.code
...
call suma Pascal, w1, w2
```
;wywołanie procedury z parametrami dyrektywą CALL
```
pop wynik_dodawania
```
;zdjęcie zwracanej prze funkcję wartości ze stosu
```
...
```

Makra

Makro to stała symboliczna przydzielana pewnemu ciągowi instrukcji. Po zdefiniowaniu makro może być użyte dowolną liczbę razy, a jego wywołanie powoduje wstawienie do kodu programu instrukcji zawartych w makrze. Makro jest zazwyczaj wykonywane szybciej od procedury, która zawiera ten sam kod. Dzieje się tak, gdyż użycie procedury jest związane ze skokiem do kodu procedury i powrotem do miejsca skąd została wywołana (CALL i RET). Wadą częstego stosowania dużych makr jest zwiększenie objętości programu, ponieważ każde wywołanie makra oznacza wstawienie jego kodu.

Makroinstrukcję można definiować w dowolnym miejscu kodu źródłowego przy użyciu dyrektyw MACRO i ENDM. Składnia deklaracji jest następująca::

```
nazwa MACRO lista_parametrów
...
kod makra
...
ENDM
```

Przykładowa makroinstrukcja:

```
mPrint MACRO tekst
MOV AH,09H
MOV DX,OFFSET tekst
INT 21H
ENDM
```

Jak widać makro wyświetla tekst w konsoli podany w zmiennej "tekst". Wywołanie tego makra może wyglądać następująco:

```
mPrint Napis1
mPrint Napis2
...
Napis1 DB 'Brak wymaganego pliku data.txt!$'
Napis2 DB 'Błąd obsługi błędów :-)$'

...
```

Jedną z zalet stosowania makr jest wbudowana obsługa parametrów. Do oznaczania wymaganych parametrów używany jest kwalifikator REQ, np.:

```
mKursorPoz MACRO wiersz:REQ, kolumna:=<1>
mov ah,02h
mov dl,kolumna
mov dh,wiersz
mov bh,0
int 10h
ENDM
```

Makro wymaga podania parametru "wiersz" natomiast parametrowi "kolumna" przypisuje wartość początkową 1. Domyślna wartość parametru musi się znajdować między znakami < >.

Jeżeli do parametrów nie zostaną przypisane wartości początkowe i nie są to parametry wymagane (kwalifikator REQ), to podczas pisania makroinstrukcji zawierających parametry opcjonalne zachodzi konieczność sprawdzenia czy argument został podany, czy też nie. Do tego celu służą następujące dyrektywy warunkowego asemblowania:

IF *wyrażenie*
Pozwala na asemblowanie jeśli wartość *wyrażenia* jest prawdziwa. Dostępne są następujące operatory relacyjne: LT, GT, EQ, NE, LE i GE.
IFE *wyrażenie*
Pozwala na asemblowanie jeśli wartość *wyrażenia* jest fałszem (wartość zerowa).
IFB <*argument*>

Pozwala na asemblowanie jeśli *argument* jest pusty (nazwa argumentu musi być ujęta w znaki < i >).

IFNB <*argument*>

Pozwala na asemblowanie jeśli *argument* nie jest pusty (nazwa argumentu musi być ujęta w znaki < i >).

IFIDN <*argument1*>, <*argument2*>

Pozwala na asemblowanie jeśli dwa argumenty są sobie równe (niezależnie od wielkości znaków).

IFDIF <*argument1*>, <*argument2*>

Pozwala na asemblowanie jeśli dwa argumenty nie są sobie równe (niezależnie od wielkości znaków).

IFDEF *nazwa*

Pozwala na asemblowanie jeśli *nazwa* została zdefiniowana.

IFNDEF *nazwa*

Pozwala na asemblowanie jeśli *nazwa* nie została zdefiniowana.

ELSE

Pozwala na asemblowanie wszystkich instrukcji aż do dyrektywy ENDIF jeśli warunek podany przez wcześniejszą dyrektywę IF był fałszywy.

ENDIF

Kończy blok poleceń rozpoczęty przez jedną z dyrektyw warunkowego asemblowania.

EXITM

Przerwanie asemblowania aktualnego makra.

Przykładowa makroinstrukcja mDELAY z użyciem dyrektyw warunkowego asemblowania:

```
mDelay MACRO val
LOCAL czekaj PUSH CX
 IFB <val>
  MOV CX,1000
 ENDIF
 IFNB <val>
  MOV CX,val
 ENDIF
czekaj:
LOOP czekaj
POP CX
ENDM
```

Jeśli parametr "val" nie zostanie podany w odwołaniu do makra, do CX zostanie przeniesiona wartość 1000. Jeżeli "val" nie okaże się parametrem pustym - jego wartość zostanie umieszczona w CX i tyle też razy wykona się pętla LOOP. Dyrektywa LOCAL dla etykiety powoduje tworzenie odrębnej nazwy dla każdego egzemplarza makra wywołanego przez ten sam program ze względu na możliwość wystąpienia błędu składni.

Turbo Assembler oferuje również *operatory makra* dzięki którym mogą one być jeszcze bardziej rozbudowane. Dostępne operatory:

&
operator podstawienia - pozwala na zastąpienie parametru przez wartość przekazaną jako argument.
< >
operator dosłownego tekstu - łączy ciąg znaków i uniemożliwia interpretację elementów listy jako oddzielnych parametrów.
!
operator dosłownego znaku - wymusza traktowanie znaku specjalnego jako znaku a nie operatora.
%
operator wyrażenia - umożliwia przekazanie obliczonego wyniku wyrażenia jako argumentu makra.
;;
komentarz makra

Dyrektywy powtórzeniowe REPT, IRP oraz IRPC umożliwiają powtórzenie jednego lub kilku poleceń. Składnia REPT jest następująca:

```
REPT licznik
   instrukcje
ENDM
```

Wartość wyrażenia "licznik" decyduje o liczbie powtórzeń (16-bitowa liczba bez znaku).

Dyrektywa IRP powtarza blok poleceń przy czym za każdym powtórzeniem używa innych wartości parametru:

```
IRP parametr,<argument1> [,argument2] ...
   instrukcje
ENDM
```

Dyrektywa IRPC działa podobnie do IRP, ale o liczbie powtórzeń decyduje liczba znaków w ciągu argumentu:

```
IRPC parametr, ciąg
   instrukcje
ENDM
```

Jeżeli ciąg zawiera znaki specjalne (spacje, przecinki, etc.), ciąg musi być umieszczony pomiędzy znakami <>.

Deklaracje danych

Zmienne

W asemblerze, jak i w innych językach programowania, mamy do dyspozycji kilka dyrektyw, które umożliwiają definiowanie zmiennych. Rozróżniamy trzy rodzaje zmiennych:

1. Zmienne pojedyncze (mają tylko jedną, pojedynczą wartość)
2. Zmienne tablicowe (struktury złożone z pól mających pojedynczą wartość)
3. Zmienne łańcuchowe (ciągi wartości jednego typu tj. ciągi liczb, napisy itp.)

Do deklarowania zmiennych używa się następujących dyrektyw:

DB (define byte) - obszar o rozmiarze jednego bajta
DW (define word) - zmienna o rozmiarze jednego słowa (dwóch bajtów)
DD (define doubleword) - zmienna o rozmiarze dwóch słów (cztery bajty)
DF - zmienna o rozmiarze trzech słów
DQ - zmienna o rozmiarze czterech słów
DT - zmienna o rozmiarze pięciu słów

Przykładowe deklarowanie zmiennych w segmencie danych:

```
X DW 0
LICZBA DW 123H
A DD ?
LITERA DB "A"
```

Deklarując zmienne nadajemy im zazwyczaj konkretne wartości. Można również przypisać literę, co w rezultacie jest przypisaniem liczby o kodzie ASCII danej litery. Jeśli wartość liczby nie jest wiadoma wpisujemy znak zapytania - wówczas zmienna nie ma określonej wartości.

W zmiennych można przetrzymywać również etykiety, np.:

```
START:
...
...
```

`ADRES16 DW START`; zmienna zawiera przesunięcie (offset) etykiety START

`ADRES32 DD START`; zmienna zawiera segment i przesunięcie etykiety START

W Asemblerze można również deklarować sekwencje bajtów, np.:

```
POTEGI_LICZBY_2 DW 1, 2, 4, 8, 16, 32, 64
KODY DB 123, 54, 12, 143
```

Można deklarować również znaki, lub znaki i liczby, np.:

```
NAPIS DB 'ciąg znaków'
KOMUNIKAT DB 'Wystąpił nieokreślony błąd :-)'
COS_TAM DB 12,'123',23,56
```

Znaki można podawać w cudzysłowach bądź apostrofach.

Tablice deklarujemy instrukcją DUP (duplikat, powielenie). Format tablicy:

```
[nazwa]    [jednostka    pamięci]    [ile    razy]
[duplikowana wartość w nawiasach ()]</b
```

`TABLICA DB 1024 DUP (?)` ;tablica o rozmiarze 1 KB, wypełniona nieokreślonymi wartościami

Można jednak nadać jej wartość, np.:

```
TABLICA DB 1024 DUP (0) ;tablica o tym samym rozmiarze lecz wyzerowana
```

Jeśli chcemy zadeklarować bardzo dużą tablicę, to deklarujemy ją w osobnym segmencie, np. jeśli będziemy przechowywać w niej obrazek, to zrobimy tak:

```
SCREEN SEGMENT
EKRAN DB 64000 DUP (?)
SCREEN ENDS
```

Stałe

Podczas pisania programów często zachodzi potrzeba użycia tych samych wartości. Wówczas można stworzyć stałą przy pomocy dyrektywy EQU. Działanie jej polega na przypisaniu nazwie stałej i niezmiennej wartości, np.:

```
LICZBA EQU 123
```

Można również przypisywać do stałych wyrażenia, np.:

```
CREEPY EQU MOV AL, DS:[SI]
```

Teraz w programie wpisanie CREEPY powoduje wykonanie instrukcji MOV AL,DS:[SI]

W Asemblerze można również przypisać inną nazwę rejestrowi, np.:

```
AKUMULATOR EQU AX
```

Innym sposobem na przydzielenie nazwy symbolicznej dla stałych i literałów jest użycie *dyrektywy znaku równości*. Stała może być zdefiniowana w czasie startu programu i zmieniana w późniejszym czasie. Deklaracja odbywa się w następujący sposób:

```
nazwa = wyrażenie
```

Jeśli chcemy zmienić wartość wyrażenia piszemy w innym miejscu programu:

```
nazwa = wyrażenie1
```

Istnieje również możliwość przydzielenia pewnej sekwencji znaków do nazwy symbolicznej. Służy temu dyrektywa TEXTEQU (tylko MASM). Ma ona następującą składnię:

```
nazwa TEXTEQU <tekst>
nazwa TEXTEQU makro_tekstowe
```

Dzięki TEXTEQU możliwe jest tworzenie aliasów reprezentujących inne zdefiniowane symbole.

Struktury

Struktura to pewne zgrupowanie zmiennych niekoniecznie tego samego typu. Zdefiniowana struktura może być nałożona na pewien obszar pamięci. Struktur nie można zagnieżdżać. Poszczególne elementy struktury są zwane *polami*. W programie struktura musi być zdefiniowana przed jej użyciem, definicja odbywa się przy użyciu dyrektywy *STRUC* w podany poniżej sposób:

```
nazwa_struktury STRUCT
pole1
pole2
...
nazwa_struktury ENDS
```

Poszczególne pola struktury mają następujący format:

```
nazwa_pola DB wartość_początkowa
```

Pod wyrażeniem nazwa_pola kryje się przesunięcie danego pola liczone względem początku struktury.

Sama definicja struktury nie powoduje włączenia danych do programu. Zapamiętywana jest ona dopiero przy zadeklarowaniu zmiennej struktury.

Ogólny format deklaracji zmiennej struktury jest następujący:

```
[nazwa] nazwa_struktury <[wart._pola1],...>
```

Przykład pracy ze strukturami:

```
PRACOWNIK STRUCT ;definicja struktury
NAZWISKO DB ''
IMIE DB ''
WIEK DB ?
STAZ DB ?
PRACOWNIK ENDS

KOWALSKI     PRACOWNIK     <'Kowalski','Jan',40,15>
```
;deklaracja struktury KOWALSKI

Mając już zadeklarowaną strukturę odwołujemy się do niej np. tak:

```
MOV AL,KOWALSKI.STAZ
```

Do AL zostanie załadowany staż przykładowego Kowalskiego. Można to również zrobić w inny sposób:

```
LEA BX,KOWALSKI
...
MOV AL,[BX].STAZ
```

Rekordy

Rekord w asemblerze to opis grupy bitów operandzie bajta lub słowa i nie powinna być mylona ze znaczeniem słowa *rekord* w językach wysokiego poziomu. Definicja rekordu przebiega według następującego schematu:

```
nazwa_rekordu     RECORD     nazwa_pola:szerokosc
[wyrazenie],...
```

Utworzony rekord to pewien rodzaj struktury danych lecz operujący na bitach. Nazwa_rekordu identyfikuje nasz rekord. Nazwa_pola określa pole w rekordzie, szerokość, liczbę bitów, z których się składa pole, a opcjonalne "wyrazenie", nadaje początkową wartość, np.:

```
DATA RECORD ROK:7,MIESIAC:4,DZIEN:5
```

Tak zadeklarowany rekord tworzy strukturę w następującej postaci:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ROK							MIESIAC				DZIEN				

Format deklaracji:

```
[nazwa] nazwa_struktury <[wartosc],..>
```

np.:

```
DZIS DATA <02,7,20>
```

Z tak zdefiniowanej daty wyizolujemy wartość roku. Do tego celu użyjemy operatora MASK. Operator ten tworzy maskę bitową - wszystkie bity odpowiadające poszczególnym pozycjom pól są ustawiane, natomiast pozostałe bity są kasowane, np.:

```
MOV BX,DZIS

AND BX,MASK ROK
;izoluje bity roku
MOV CL,ROK
;liczba bitów z prawej strony pola
SHR BX,CL
;w BX mamy naszą datę
```


Operatory asemblera

Operatory asemblera to szczególne typy wyrażeń, które operują na liczbach bądź danych i zwracają określone wartości.

Operatory arytmetyczne

Asembler obsługuje następujące operatory arytmetyczne: **+**, **-**, ***, **/**, które oznaczają dodawanie, odejmowanie, mnożenie oraz dzielenie liczb całkowitych.

Operator PTR

Operator ten wymusza zmianę domyślnej wielkości operandu. Jest używany w przypadku poleceń, w których wielkość operandu jest niejasna. Operand ten musi być użyty ze standardowymi typami asemblera: BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, TBYTE, NEAR lub FAR w następującej formie:

```
typ PTR wyrażenie
```

np.:

```
MOV BYTE PTR [BX],5
CALL FAR PTR PROC1
```

Operator SEG

Zwraca wartość segmentu będącego częścią adresu etykiety lub zmiennej np.:

```
MOV DX,SEG Napis ;kopiuje do DX segment zmiennej Napis
```

Operator OFFSET

Podaje wartość przesunięcia (offsetu) względem początku segmentu, w którym znajduje się dane wyrażenie. Operand docelowy musi być 16-bitowym rejestrem np.:

```
MOV DX,OFFSET Napis
```

Operator HIGH

Zwraca bardziej znaczące (górne) 8 bitów danej liczby lub stałego wyrażenia, np.:

```
MOV AL,HIGH LICZBA
```

Operator LOW

Zwraca mniej znaczące (dolne) 8 bitów stałego wyrażenia lub liczby, np.:

```
SUB CL,LOW LICZBA
```

Operator TYPE

Zwraca liczbę całkowitą reprezentującą typ danego wyrażenia. Jeśli jest ona typu BYTE, to 2, WORD, to 4, itd. Jeśli wyrażenie jest etykietą typu NEAR zwraca 0FFFFH, a jeśli FAR 0FFFEH, np.:

```
MOV AX,TYPE DANE
```

```
CALL (TYPE TYP_ADRESU) PTR PROCEDURA
```

Operator LENGTH

Zwraca liczbę elementów (nie bajtów) zadeklarowanych w definicji danej zmiennej, np.:

```
TAB1 DB 24 DUP (4)
TAB2 DW 75 DUP (85 DUP (0))
...
MOV CX,LENGTH TAB1
;w CX będzie 24
SUB BX,LENGTH TAB2
;BX zostanie zmniejszony o 75
```

Operator WIDTH

Zwraca liczbę bitów określonego pola w zmiennej, która została zadeklarowana przy użyciu dyrektywy RECORD.

Operator SIZE

Zwraca łączną liczbę bajtów alokowanych dla zmiennej (inaczej SIZE = LENGTH * TYPE).

Operator SHORT

Ustawia typ danej etykiety na SHORT. Zazwyczaj jest wykorzystywany w instrukcjach skoku, np.:

```
JMP SHORT ETYKIETA1
```

Operator MOD

Jest to operator modułu - zwraca resztę z dzielenia

Operator MASK

Podaje maskę dla pozycji bitowych zajmowanych w rekordzie przez pole. Bity odpowiadające danemu polu są równe 1, a pozostałe 0. Zmienna musi być zadeklarowana z użyciem dyrektywy RECORD.

Operator (.)

Nazwa występująca po znaku kropki (.) identyfikuje pole znajdujące się w zdefiniowanej strukturze. Identyfikacja odbywa się poprzez dodanie przesunięcia pola do przesunięcia zmiennej. Stosowany format:

```
zmienna.pole
```

Operatory relacji

- EQ - zwraca prawdę gdy argumenty są równe
- NE - zwraca prawdę gdy argumenty są różne
- LT - zwraca prawdę gdy argument prawy jest większy niż lewy
- GT - zwraca prawdę gdy argument lewy jest większy niż prawy
- LE - zwraca prawdę gdy argument lewy jest niewiększy niż prawy
- GE - zwraca prawdę gdy argument prawy jest niemniejszy niż lewy

`MOV CX,(warunek EQ 100)` ;zwróci prawdę do CX kiedy warunek będzie równy 100