# Definicja podstawy kodu w Assemblerze

Dyrektywa .model odpowiada za wybranie modelu pamięci dla programu który jest tworzony w języku asembler. Modele charakteryzują się różnymi ograniczeniami odnośnie maksymalnej przestrzeni dla kodu i danych oraz dostępu do pod-procedur i danych. Wybór odpowiedniego modelu jest kompromisem miedzy wydajności i wielkością programu.

| .model | Właściwości |
|---|---|
| Tiny | Segment kodu + Segment danych <= 64 kB |
| Small | Segment kodu <= 64 kB \| segment danych <= 64 kB \| 1 segment kodu \| 1 segment danych |
| Medium | Segment kodu – dowolna wielkość \| Segment danych <= 64 kB \| wiele segmentów kodu \| 1 segment danych |
| Compact | Segment kodu <= 64 kB \| Segment danych – dowolna wielkość \| 1 segment kodu \|wiele segmentów danych |
| Large | Segment kodu > 64 kB \| segment danych > 64 kB \| wiele segmentów kodu \| wiele segmentów danych |
| Huge | Model Large + poszczególne zmienne (np. tablice) > 64 kB |

---

.stack / org 100h – linia ta mówi kompilatorowi, że nasz kod będzie znajdował się pod adresem 100h (256 dziesiętnie) w swoim segmencie. Jest to typowe dla programów .com. DOS, uruchamiając taki program, szuka wolnego segmentu i kod programu umieszcza dopiero pod adresem (czasami zwanym offsetem – przesunięciem) 100h.

---

.data - segment z danymi służący do ich deklaracji

Dyrektywa .data ? służy do deklarowania niezainicjalizowanych danych. Podczas definiowania dużej ilości niezainicjalizowanych danych ta dyrektywa sprawia, że program będzie miał mniejszy rozmiar

## *ZMIENNE*

Zmienną w asemblerze nazywamy miejsce w pamięci oznaczone przez etykietę (nazwa zmiennej). Do dyspozycji są następujące dyrektywy umożliwiające definiowanie zmiennych :

DB (define byte) – pozwala zdefiniować zmienna (obszar pamięci) o rozmiarze jednego bajta.
DW (define word) – pozwala zdefiniować zmienna o rozmiarze dwóch bajtów, czyli słowa.
DD (define doubleword) – definiuje zmienna o rozmiarze dwóch słów czyli czterech bajtów.

Dodatkowo wykorzystywane są dyrektywy definiujące większe pola:

DF – definiuje zmienna o rozmiarze trzech słów (6 bajtów)
DQ – definiuje zmienna o rozmiarze czterech słów (8 bajtów)
DT – definiuje zmienna o rozmiarze pięciu słów (10 bajtów)

* Użycie znaku ? powoduje, że kompilator nie przypisuje zmiennej wartości początkowej – rezerwuje jedynie pamięć.

## *JAK OBSŁUGIWAĆ WARTOŚCI TEKSTOWE*

Funkcja ah=9 przerwania DOSa - najbardziej popularna i uniwersalna funkcja do wyświetlania łańcucha znaków – wymagane jest jedynie zakończenie go $. Przykład poniżej:

```
Mov     ah, 9
mov     dx, offset tekst ; NASM/FASM: bez "offset"
int     21h
 ...
String  db "String$"
```

Funkcja ah=0E przerwania 10h (wyświetla po jednym znaku):J

```
mov   ah, 0eh
mov   al, 'a'
int   10h
```

Funkcja ah=2 przerwania DOSa także wyświetla po jednym znaku:

```
mov   ah, 2
mov   dl, 'a'
int   21h
```

## *INSTRUKCJE KODU*

**.code** - Wskazuje początek segmentu gdzie znajduje się kod programu.

---

Instrukcja **MOV** kopiuje dane ze źródła do miejsca docelowego. Format tej instrukcji wygląda następująco:

MOV cel, źródło

Instrukcja MOV stawia pewne wymagania:

- -Oba operandy muszą być tego samego rodzaju
- -Oba operandy nie mogą być jednocześnie adresem w pamięci
- -Rejestr wskaźnika instrukcji nie może być rejestrem docelowym

Dozwolone kombinacje można zapisać w następujący sposób:

```
MOV rejestr, rejestr
MOV pamięć, rejestr
MOV rejestr, pamięć
MOV pamięć, stała
MOV rejestr, stała
```

## *OPERACJE ARYTMETYCZNE*

**DODAWANIE** – dodawać możemy 8, 16 lub 32 bitowe wartości ze znakiem lub bez.

Instrukcja **add** posiada parametry jak w instrukcji mov. Przykład:

```
Mov ax, 10
mov bx, 20
add ax, bx  ; ax = 30
```

Instrukcja **adc** posiada parametry jak w instrukcji add z dodatkiem znacznika przeniesienia (CF) – znacznik ten jest dodawany do wyniku.

Instrukcja **inc** dodaje 1 do argumentu, używana głównie do sterowania licznikiem pętli.

**ODEJMOWANIE** – tak jak w przypadku dodawania operację tą możemy wykonywać na 8, 16, 32 bitowych liczbach.

Instrukcja **sub** posiada parametry jak w instrukcji add, przykład:

```
Mov ax, 10
mov bx, ax
sub ax, bx  ; bx = 10
```

Instrukcja **sbb** działa analogicznie do adc czyli wykonuje odejmowanie z uwzględnieniem znacznika przeniesienia (CF)

Instrukcja **dec** odejmuje 1 od argumentu, używana podobnie jak inc przy licznikach pętli.

**MNOŻENIE** – możemy je wykonywać na bajtach, słowach lub podwójnych słowach. Są dostępne dwie instrukcje dla mnożenia mul dla argumentów bez znaku oraz imul dla argumentów ze znakiem. Wynikiem mnożenia dwóch bajtów jest słowo (AH:AL), dwóch słów podwójne słowo (DX:AX), natomiast dwóch wartości 32-bitowych poczwórne słowo (EDX:EAX)

W podstawowej postaci, jedynym parametrem instrukcji mnożenia jest mnożnik (liczba przez którą mnożymy), przekazany w rejestrze lub komórce pamięci. Mnożenie wykonywane jest:

W przypadku liczb 8-bitowych na rejestrze AL, wynik jest wstawiany do AX
W przypadku liczb 16-bitowych na rejestrze AX wynik jest wstawiany do DX:AX
W przypadku liczb 32-bitowych na rejestrze EAX, wynik jest wstawiany do EDX:EAX

**DZIELENIE** – podobnie jak w przypadku mnożenia, mamy do dyspozycji dwie instrukcje (div oraz idiv). Parametrem instrukcji dzielenia jest dzielnik przekazany w rejestrze lub komórce pamięci.

Operacja dzielenia jest wykonywana:
- dla argumentu 8 bitowego na rejestrze AX, wynik dzielenia (iloraz) jest umieszczany w AL, a reszta z dzielenia w AH
- dla argumentu 16 bitowego na rejestrach DX:AX (32-bitowa liczba). Wynik dzielenia (iloraz) jest umieszczany w rejestrze AX, a reszta w DX
- dla argumentu 32 bitowego na rejestrach EDX:EAX (liczba 64 bitowa). Wynik dzielenia (iloraz) jest umieszczany w EAX natomiast reszta w EDX

## *OPERACJE LOGICZNE*

Parametry operacji logicznych sa takie same jak w przypadku instrukcji mov.

**AND** – mnożenie logiczne. Wykonuje logiczne „i" na bitach argumentów i umieszcza wynik w pierwszym z tych argumentów(nie może to być stała).

**OR** – dodawanie logiczne. Wykonuje operację „lub" na bitach argumentów. Wynik umieszcza w pierwszym argumencie (nie może to być stała).

**XOR** – logiczne „albo" na bitach argumentów. Wynik operacji umieszczany jest w pierwszym argumencie (nie może to być stała). Oto przykład operacji XOR:

```
mov al, 10011011b
mov bl, 11001100b
xor al, bl              ; al = 01010111b
```

**NOT** – negacja logiczna bitów. Jest to instrukcja jednoargumentowa. Odwraca (zamienia 0->1 i 1->0) bity argumentu i zostawia wynik w argumencie.

**TEST** – operacja „i" bez zmiany wartości argumentów. Jej działanie jest identyczne z działaniem instrukcji AND, jedyna różnica polega na tym, że wynik operacji nie jest nigdzie umieszczany.

## *ADRESOWANIE ZMIENNYCH*

Aby odnieść się do konkretnego adresu w pamięci możemy użyć tak jak to robiliśmy do tej pory nazwy symbolicznej lub też konkretnej liczby albo adresu zawartego w dowolnym rejestrze. Aby oświadczyć, że dana wartość ma być traktowana jako offset (przesunięcie względem początku segmentu) musimy umieścić ją między nawiasami kwadratowymi.

Pobranie adresu zmiennych następuje poprzez komendę:
mov dx, offset ….

## *TABLICE*

Aby zdefiniować tablicę w segmencie danych inicjalizowanych .**data** używamy dyrektyw db, dw itd. Możemy wykorzystać dyrektywę TIMES by nie pisać wielokrotnie tego samego.
Aby zdefiniować tablicę w segmencie danych nieinicjalizowanych **.bss** używamy dyrektyw resb czy resw.

```
segment .data

a1 dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10     ; tablica 10 podwójnych słów
zainicjalizowana na 1,2,..,10

a2 dw 0, 0, 0, 0, 0, 0, 0, 0, 0, 0      ; tablica 10
słów zainicjalizowana na 0

a3 times 10 dw 0                             ; j.w. z wykorzystaniem TIMES

a4 times 200 db 0                            ; tablica bajtów zawierająca
200 zer i potem 100 jedynek
      times 100 db 1

segment .bss

a5 resd 10                                ; tablica 10 podwójnych słów
a6 resw 100                              ; tablica 100 słów
```

Aby uzyskać dostęp do elementu tablicy, należy obliczyć jego adres.
Jeżeli chcemy obliczyć adres elementu tablicy (np. w celu przekazania go do fukcji) możemy użyć instrukcji lea

```
lea rdx, [rbx + 2*rcx - 2]      ;  jeżeli rcx=2 to rdx będzie zawierał
                                ;  adres drugiego elementu tablicy (a nie jego
wartość)
```

## *STOS – jak używać?*

Rozkaz MOV może być użyty w celu uzyskania dostępu do stosu poprzez odpowiednie zaadresowanie pamięci, która używa rejestru BP jako bazowego wskaźnika określającego adres względny wewnątrz segmentu stosu. Na przykład:
MOV AX,[BP+4] ; ładowanie rejestru AX zawartości słowa spod adresu względnego BP+4, w segmencie SS. Najczęściej jednak dostęp do stosu realizowany jest za pomocą rozkazów PUSH i POP. PUSH ładuje argument na wierzchołek stosu, POP zdejmuje argument z wierzchołka stosu.

Przykład:

```
MOV AX, 6
PUSH AX      ; odłóż na stos wartość 6
POP BX       ; zdejmij argument z wierzchołka stosu (tu: 6) i zapamiętaj go w
             ; rejestrze BX
```

## *ETYKIETY – SKOKI BEZWARUNKOWE I WARUNKOWE*

Etykiety istnieją jedynie na poziomie kodu asemblerowego. Po asemblacji nie pozostaje po nich żaden ślad. Służą do nazywania konkretnych adresów w pamięci
Gdy chcemy przeskoczyć w wykonaniu naszego programu do naszej funkcji posługujemy się nazwą etykiety jako argumentem dla odpowiedniej instrukcji modyfikującej rejestr EIP i ew. rejestr CS tak aby razem wskazywały na początek funkcji.

Instrukcja *jmp* modyfikuje wartość rejestru EIP, zależnie od przekazanego parametru. Istnieją 3 odmiany tej instrukcji:

- **jmp short** - korzystamy z niej, gdy procedura, którą chcemy wywołać jest oddalona o 128 bajtów w tył lub do przodu; najszybsza instrukcja z rodziny.
- **jmp near** - korzystamy z niej, gdy procedura, którą chcemy wywołać jest w tym samym segmencie co procedura wywołująca (w przypadku płaskiego modelu pamięci - w tym samym programie).
- **jmp far** - korzystamy z niej, gdy procedura, którą chcemy wywołać jest w innym segmencie, zmianie ulega również rejestr CS; w przypadku płaskiego modelu pamięci, korzystamy z niej gdy wywoływana funkcja znajduje się poza naszym procesem.

Instrukcja *call* działa identycznie do instrukcji jmp z tą różnicą, że przed przeskokiem układa na stosie bieżące wartości rejestrów EIP oraz ew. CS tak aby później było można wrócić do miejsca gdzie wykonany był skok przy użyciu którejś instrukcji z rodziny ret. Istnieją dwie główne odmiany instrukcji call:

- **call near** - korzystamy z niej, gdy wywoływana procedura jest w tym samym segmencie kodu (w przypadku płaskiego modelu pamięci odnosi się to do tego samego procesu)
- **call far** - korzystamy z niej, gdy wywoływana procedura jest poza naszym procesem.

Instrukcja **cmp** (compare) odejmuje od siebie argumenty (od drugiego pierwszy) i na tej podstawie ustawia znaczniki (wynik odejmowania nie jest nigdzie składowany.

Skoki warunkowe opierają się w swoim działaniu na stanie rejestru flag. Stany poszczególnych jego bitów określają bezpośrednio czy dany skok ma być wykonany czy też nie.

Zestaw instrukcji skoków warunkowych poniżej:

| warunek | | działanie | negacja warunku | |
|---|---|---|---|---|
| je (lub jz) | = | jump if equal | jne (lub jnz) | ! = |
| bez znaku | | | | |
| ja | > | jump if above | jna | ! > |
| jae | >= | jump if above or equal | jnae | ! >= |
| jb | < | jump if below | jnb | ! < |
| jbe | <= | jump if below or equal | jnbe | ! <= |
| ze znakiem | | | | |
| jg | > | jump if greater | jng | ! > |
| jge | >= | jump if greater or equal | jnge | ! >= |
| jl | < | jump if less | jnl | ! < |
| jle | <= | jump if less or equal | jnle | ! <= |

Skoki warunkowe mogą być także wykonywane bezpośrednio testując jedną z flag. Mogą to być flagi CF, SF, OF i PF. Instrukcje im odpowiadające to:

| instruckja | działanie | instrukcja z negacją warunku |
|---|---|---|
| jc | jump if CF=1 | jnc |
| jz | jump if ZF=1 | jnz |
| js | jump if SF=1 | jns |
| jo | jump if OF=1 | jno |
| jp | jump if PF=1 | jnp |

## *PĘTLE*

**LOOP**

Instrukcja LOOP jest podstawową instrukcją realizującą pętle. Jako licznik pętli wykorzystywany jest rejestr CX. Parametrem instrukcji LOOP jest etykieta pętli (podana przez nazwę lub bezpośrednio przez adres). Najprostsza pojedyncza pętla LOOP wygląda następująco :

```
      Mov cx, rozmiar pętli
etykieta:

      < instrukcje pętli >
      loop etykieta
```

Instrukcja LOOP zmniejsza CX o 1 jeśli CX jest różne od zera, wykonuje skok do wskazanej jako parametr etykiety pętli.
Najdłużej wykonuje się pętla o rozmiarze 0. Wynika to z kolejności wykonywania operacji przez instrukcje LOOP: najpierw odejmuje 1 od licznika (czyli z CX równego 0 otrzymujemy CX równe 65536) i dopiero wtedy sprawdza czy licznik jest różny od zera.
Instrukcja JCXZ ominie pętle gdy CX będzie równe zero.
Jeśli trzeba w pętli użyć rejestru CX to najlepszym sposobem do tego jest wykorzystanie stosu do zapamiętania każdorazowo stanu tego rejestru (stosowane również do realizacji pętli zagnieżdżonych).

LOOPNE, LOOPZ – instrukcje działają tak samo (można ich używać zamiennie):
- odejmują 1 od CX
- jeśli CX jest różne od zera i ZF =0, to wykonują skok do podanej etykiety.

## *OPERACJE NA ŁAŃCUCHACH*

Instrukcja REP powtarza CX razy (zawartość CX) instrukcję po niej następującą: LODS, MOVS, STOS, CMPS, SCAS, OUTS.

LODS – odczytanie elementu z łańcucha
MOVS – kopiowanie łańcuchów
STOS – wypełnienie łańcucha wartością
CMPS – porównanie dwóch łańcuchów
SCAS – przeszukanie łańcucha (wyszukiwanie określonej wartości).

## *LITTLE ENDIAN CZYLI BAJTY W ASEMBLERZE*

W architekturze x86 formą zapisu bajtów jest Little Endian.
Oznacza to, że wielobajtowe wartości są zapisane w kolejności od najmniej do najbardziej znaczącego (patrząc od lewej strony), bardziej znaczące bajty będą miały "wyższe" (rosnące) adresy.
Należy mieć na uwadze, że odwrócona zostaje kolejnośc bajtów a nie bitów.
Zatem 32-bitowa wartość B3B2B1B0 mogłaby na procesorze z rodziny x86 być zaprezentowana w ten sposób:

Reprezentacja kolejności typu little-endian
Byte 0 Byte 1 Byte 2 Byte 3