LEKSYKON KIESZONKOWY

ASEMBLER

Poznaj Asemblera i dołącz do elity programistów

Dowiedz się, jak jest zbudowany i jak działa procesor

Poznaj język Asembler i narzędzia umożliwiające tworzenie w nim programów

Naucz się pisać wydajny kod działający w systemach DOS i Windows

DAWID FARBANIEC

Helion

Redaktor prowadzący: Michał Mrowiec

Projekt okładki: Maciek Pasek

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?asemlk
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:
ftp://ftp.helion.pl/przyklady/asemlk.zip

ISBN: 978-83-246-4347-9

Printed in Poland.

Spis treści

Rozdział 4. Praca z danymi

W tym rozdziale nauczysz się, jak pracować z danymi, takimi jak na przykład zmienne czy struktury. Dowiesz się też, czym jest adres danych w pamięci i jak go pobierać.

4.1. Adresowanie i wskaźniki

W Asemblerze bardzo ważne jest to, aby wiedzieć, czym różni się adres zmiennej (`OFFSET`) od jej wartości. Adres to miejsce, gdzie znajduje się zmienna, natomiast wartością są dane umieszczone pod tym adresem.

Operator OFFSET

Operator `OFFSET` służy do pobierania adresu zmiennej.

Oto przykład jego użycia:

```
.386
.model   flat, stdcall
option   casemap: none


include      windows.inc
include      kernel32.inc
include      user32.inc

includelib   user32.lib
includelib   kernel32.lib

.data
zmienna db "Witaj!", 0

.code
start:
   mov eax, offset zmienna ; do rejestru EAX skopiuj adres zmiennej
   invoke MessageBox, NULL, eax, NULL, NULL ; wyświetl komunikat
   invoke ExitProcess, NULL ; koniec programu
end start
```

Operatora `OFFSET` używamy również, gdy pobieramy adres zmiennej, odkładając ją na stos przed wywołaniem procedury, na przykład:

```
push NULL
push NULL
push offset zmienna ; pobieramy adres zmiennej i odkładamy na stos
push NULL
call MessageBox
```

Warto zaznaczyć, że operator OFFSET nie działa na zmiennych lokalnych.

Zmienne lokalne

Zmienne lokalne możemy zadeklarować na przykład wewnątrz jakiejś procedury. Aby to zrobić, podajemy najpierw słowo LOCAL, dalej nazwę zmiennej, a po dwukropku jej rodzaj.

Na przykład:

```
myProc proc
   LOCAL zmienna1: DWORD ; deklaracja zmiennej lokalnej o rozmiarze podwójnego słowa
      ; jakiś kod
   Ret ; powrót
myProc EndP
```

Operator ADDR

Operatora ADDR używamy najczęściej przy przekazywaniu argumentów do funkcji wywoływanej za pomocą składni INVOKE, na przykład:

```
invoke MessageBox, NULL, addr MsgText, addr MsgCaption, NULL
```

Dereferencja (operator [ ])

Posłużymy się tutaj prostym przykładem:

```
mov eax, lpvar     ; skopiuj adres do EAX
mov eax, [eax]     ; dereferencja
mov nuvar, eax     ; skopiuj EAX do nowej zmiennej
```

Nawiasy kwadratowe służą do wydobycia wartości spod adresu w EAX. Operacja ta jest nazywana dereferencją.

Instrukcja LEA

Instrukcja ta jest skrótem angielskiego wyrażenia: *Load Effective Address*. Służy ona do pobierania adresu i kopiowania go do pierwszego operandu, na przykład:

```
lea eax, zmienna
invoke MessageBox, NULL, eax, NULL, NULL
```

4.2. Zmienne i stałe

W języku Asembler prócz operacji na rejestrach możemy definiować zmienne oraz stałe, by wykonywać na nich różne operacje.

Stałe

Stałe umieszczamy w sekcji `.CONST`.

Posłużę się tutaj prostym przykładem:

```
.const ; stałe
jeden equ 1 ; definicja stałej
osiem equ 2*4 ; wartość wyliczona
```

Zmienne

Zmienne możemy podzielić na zainicjowane jakąś wartością oraz niezainicjowane. Zmienne zainicjowane deklarujemy z wartością początkową z prawej strony, niezainicjowane natomiast mają z prawej strony znak zapytania. Zmienne zainicjowane deklarujemy w sekcji `.DATA`, a niezainicjowane w — `.DATA?`. Obie te sekcje występują przed sekcją kodu (`.CODE`).

Zmienne o rozmiarze jednego bajtu/ciągu bajtów

Zmienne te deklarujemy dyrektywą `BYTE` (lub krótko: `DB`; ang. *Define Byte*).

```
.data ; dane zainicjowane wartością podaną z prawej strony
b1 BYTE  0 ; jeden bajt
b2 db    0 ; również jeden bajt
b3 db    100 dup(0) ; sto bajtów
b4 db    "asd", 0 ; cztery bajty (ciąg znaków ASCII zakończony zerem)
b5 SBYTE 0 ; jeden bajt ze znakiem
t1 dt    0 ; dziesięć bajtów
```

Zmienne o rozmiarze jednego słowa

Zmienne te deklarujemy przy użyciu dyrektywy `WORD` (lub krótko: `DW`; ang. *Define Word*).

```
w1 word  0 ; słowo
w2 dw    0 ; również słowo
```

Zmienne o rozmiarze podwójnego słowa

Zmienne te deklarujemy za pomocą dyrektywy `DWORD` (lub krótko: `DD`; ang. *Define Double Word*).

```
d1 dword 0 ; podwójne słowo
d2 dd    0 ; również podwójne słowo
d3 sdword 0 ; podwójne słowo ze znakiem
```

Zmienne o rozmiarze poczwórnego słowa

Zmienne te deklarujemy za pomocą dyrektywy `QWORD` (lub krótko: `DQ`; ang. *Define Quad Word*).

```
q1 qword 0 ; poczwórne słowo
q2 dq    0 ; również poczwórne słowo
q3 sqword 0 ; poczwórne słowo ze znakiem
```

Zmienne o rozmiarze 6 bajtów

Zmienne te deklarujemy przy użyciu dyrektywy `FWORD` (lub krótko: `DF`; ang. *Define Fword*).

```
f1 fword 0 ; zmienna 6-bajtowa
f2 df    0 ; również zmienna 6-bajtowa
```

Zmienne używane w instrukcjach MMX i SSE

Zmienne te deklarujemy za pomocą dyrektyw: `MMWORD` i `XMMWORD`. Pierwsza ma 64 bity, a druga — 128.

```
m1 mmword 0.0 ; 64-bitowa zmienna używana w instrukcjach MMX i SSE
x1 xmmword 0.0 ; 128-bitowa zmienna używana w instrukcjach MMX i SSE
```

Zmienne do przechowywania liczb rzeczywistych

Zmienne te deklarujemy przy użyciu dyrektyw: `REAL4`, `REAL8` i `REAL10`.

```
n1 real4 0.0 ; liczba zmiennoprzecinkowa pojedynczej precyzji (4 bajty)
n2 real8 0.0 ; liczba zmiennoprzecinkowa podwójnej precyzji (8 bajtów)
n3 real10 0.0 ; liczba zmiennoprzecinkowa 10-bajtowa
```

4.3. Praca ze strukturami

W poprzednim rozdziale była mowa o zmiennych i stałych. Teraz opiszę typ danych, jakim jest struktura.

Deklaracja przykładowej struktury dla asemblera MASM wygląda następująco (deklarację tę umieszczamy PRZED SEKCJĄ `.DATA`):

```
Rect STRUCT
      left    dword  ?
      top     dword  ?
      right   dword  ?
      bottom  dword  ?
Rect ENDS
```

Najpierw pojawia się nazwa struktury (Rect), po niej słowo STRUCT, a niżej są umieszczone pola struktury. Każde z nich jest podwójnym słowem (DWORD), niezainicjowanym żadną wartością (znak ?). Na końcu mamy znów nazwę struktury i słowo kończące deklarację ENDS.

Teraz utwórzmy zmienną typu Rect. Robimy to w ten sposób:

```
.data
Rct Rect <>
```

Do pól struktury odwołujemy się, pisząc nazwę struktury i po kropce jej pole, na przykład:

```
mov Rct.left,   12
mov Rct.top,    24
mov Rct.right,  31
mov Rct.bottom, 10
```

Jeżeli jakaś funkcja wymaga od nas adresu struktury jako argumentu, używamy operatora ADDR, na przykład:

```
invoke Func1, param1, param2, addr Rct
```

Możemy również tworzyć struktury zawierające w sobie inne struktury, na przykład:

```
Rect STRUCT
      left    dword  ?
      top     dword  ?
      right   dword  ?
      bottom  dword  ?
Rect ENDS

Point STRUCT
   x dword ?
   y dword ?
Point ENDS

MyStruct STRUCT
   item1 Rect <> ; struktura Rect
   item2 Point <> ; struktura Point
MyStruct ENDS

.data
Struct1 MyStruct <>
```

Wtedy do pól odwołujemy się w następujący sposób:

```
mov Struct1.item1.left, 10
```

Do pól struktury możemy się także odnosić poprzez adres w rejestrze z dodanym przesunięciem, na przykład:

```
mov eax, lpRct ; adres struktury do EAX
mov [eax],    DWORD PTR 10
mov [eax+4],  DWORD PTR 12
mov [eax+8],  DWORD PTR 14
mov [eax+12], DWORD PTR 16
```

Można to również zrobić przy użyciu dyrektywy ASSUME:

```
ASSUME eax:PTR RECT
    mov eax, lpRct
    mov [eax].left,   10
    mov [eax].top,    12
    mov [eax].right,  14
    mov [eax].bottom, 16
ASSUME eax: nothing
```

Skorowidz

D

E

F

G

H

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

LEKSYKON KIESZONKOWY

ASEMBLER

„Asembler" to słowo wywołujące przerażenie u laików i dreszcz emocji u specjalistów. Osoby znające ten język są uważane za elitę wśród profesjonalnych programistów. Nic dziwnego — w końcu nikt lepiej niż oni nie zna podstaw działania współczesnych procesorów i technik umożliwiających tworzenie bardzo wydajnych programów komputerowych. Jednak czy wiedza o Asemblerze naprawdę jest tajemna i dostępna tylko nielicznym? Czy język ten jest aż tak trudny do opanowania? Wcale nie!

Przekonaj się o tym, sięgając po książkę *Asembler. Leksykon kieszonkowy*. Przystępnie i rzeczowo wyjaśni Ci ona sposób działania procesorów zgodnych z architekturą x86, przedstawi strukturę 16-bitowych programów dla środowiska DOS i 32-bitowych programów dla systemów operacyjnych Windows, a także poszerzy Twoją wiedzę o informacje dotyczące Asemblera dla architektury 64-bitowej. Dzięki lekturze poznasz instrukcje języka, najważniejsze techniki programistyczne, sposoby używania elementów interfejsu użytkownika oraz narzędzia niezbędne w pracy programisty.

- Podstawowe pojęcia związane z Asemblerem i przydatne narzędzia
- Wiadomości na temat architektury x86 i korzystania z rejestrów procesora
- Budowa 16-bitowych programów DOS i 32-bitowych programów Windows
- Korzystanie z makroinstrukcji oraz różnych typów danych
- Tworzenie programów w systemie Windows i używanie kontrolek UI

Naucz się wydajnie programować w Asemblerze!

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-246-4347-9

Cena 29,00 zł