

Programowanie niskopoziomowe

Laboratorium 05 – Materiały uzupełniające

„Procedury i makroinstrukcje”

Cześć materiałów autorstwa prof. A. Timofiejewa

Tworzenie i korzystanie z bibliotek

```
.586P
; The PROG2.ASM module
; Flat memory model
.MODEL FLAT, STDCALL
PUBLIC PROC1
_TEXT SEGMENT
PROC1 PROC
    MOV EAX, 1000
    RET
PROC1 ENDP
_TEXT ENDS
END
```

Asemlacja do .obj:

```
ML /coff /c PROG2.ASM
```

Tworzenie biblioteki lib1

```
LIB /OUT:LIB1.LIB PROG2.OBJ
```

Dodanie innego modułu do biblioteki

```
LIB LIB1.LIB MODUL.OBJ
```

Listowanie modułów

```
LIB/LIST LIB1.LIB
```

Usunięcie modułu z biblioteki

```
LIB/REMOVE:MODUL.OBJ LIB1.LIB
```

Przykład wykorzystania biblioteki lib1

```
.586P
; Flat memory model
.MODEL FLAT, STDCALL
;-----
; Prototype of the external procedure
EXTERN    PROC1@0:NEAR
;-----
INCLUDELIB LIB1.LIB
;-----
; Data segment
_DATA SEGMENT
_DATA ENDS
; Code segment
_TEXT SEGMENT
START:
    CALL    PROC1@0
    RET ; Exit
_TEXT ENDS
END START
```

Wyjście z programu lib: enter w pustej linii

Podprogramy

Definiowanie podprogramów

Podprogramy są definiowane za pomocą *dyrektywy PROC*.

Dyrektywa PROC w języku MASM ma następującą składnię:

```
_nazwa PROC [_odległość] [_typ_językowy] [_widoczność]
[<parametry_prologu>] [USES lista_rejestrów]
[,argument [:_typ]] ...
[LOCAL _lista_zmennych]
... instrukcje podprogramu
_nazwa ENDP
```

W pole `_nazwa` musi być nazwa podprogramu. Ta nazwa musi być powtórzona przed ostatnim słowem kluczowym ENDP.

Pole `_odległość` opisuje maksymalną odległość w bajtach między miejscem wywołania i miejscem rozmieszczenia podprogramu. Opcja może mieć wartości:

NEAR - miejsce wywołania i miejsce rozmieszczenia znajdują się w jednym segmencie,

NEAR16 - miejsce wywołania i miejsce rozmieszczenia znajdują się w jednym segmencie rozmiarem 2^{16} ,

NEAR32 - miejsce wywołania i miejsce rozmieszczenia znajdują się w jednym segmencie rozmiarem 2^{32} ,

FAR - miejsce wywołania i miejsce rozmieszczenia znajdują się w różnych segmentach,

FAR16 - miejsce wywołania i miejsce rozmieszczenia znajdują się w różnych segmentach rozmiarem 2^{16} każdy,

FAR32 - miejsce wywołania i miejsce rozmieszczenia znajdują się w różnych segmentach rozmiarem 2^{32} każdy.

Jeśli pole `_odległość` nie jest zdefiniowane, to asembler uwzględnia model pamięci w dyrektywie MODEL, a jeśli model nie jest zdefiniowany, to asembler stosuje wartość domyślną NEAR.

Opcja `_typ_językowy` może mieć wartość BASIC, C, FORTRAN, PASCAL, STDCALL, lub SYSCALL. Od wartości tego pola zależy kolejność przesyłania argumentów oraz reguły korzystania ze stosu (tab. 5.15). Typy BASIC, FORTRAN i PASCAL są faktycznie jednakowe.

Właściwości typów językowych dla podprogramów (funkcji)

Typ językowy						Właściwość
BASIC	C	FORTRAN	PASCAL	STDCALL	SYSCALL	
	+			+		Podkreślenie z przodu nazwy (w bibliotece)
+		+	+			Litery duże
+		+	+			Umieszczanie na stosie argumentów od lewa na prawo
	+			+	+	Umieszczanie na stosie argumentów od prawa na lewo
	+			Tylko dla funkcji VARARG		Wywołujący odpowiada za przesunięcie stosu
+		+	+			Zapisywanie EBP na stos
	+			+	+	Typ funkcji VARARG jest dozwolony

Pole `_widoczność` w deklaracji podprogramu może przyjmować wartości `PRIVATE`, `PUBLIC` lub `EXPORT`. Wartości `PRIVATE` i `PUBLIC` definiują, w jakim stopniu podprogram (funkcja) jest widoczny dla innych modułów. Wartość `EXPORT` definiuje widoczność podobną do `PUBLIC`, oraz powoduje, że konsolidator umieszcza adres funkcji w tablicy eksportu z modułu.

Opcja `lista_rejestrów` w operatorze `USES` jest listem rejestrów, które programista chce załadować na stos w prologu i przeczytać ze stosu w epilogu. Między nazwami rejestrów musi być spacja lub przecinek.

Pola `_argument` i `_typ` opisują formalne argumenty podprogramu i ich typ. Ostatni argument może mieć typ `VARARG`, co oznacza, że w wywołaniu podprogramu na miejscu tego argumentu formalnego może znajdować się niezdefiniowana ilość argumentów faktycznych.

Pole `<parametry_prologu>` zawiera parametry przekazywane prologu oraz opcje, które definiują, jak asembler musi wygenerować instrukcje prologu i epilogu.

Asembler MASM dodaje prolog, gdy spotka pierwszą instrukcję podprogramu, a epilog – gdy spotka instrukcję `„ret”` lub `„iret”`.

Instrukcje prologu i epilogu nie są dodawane, jeśli podprogram nie używa argumentów i nie korzysta ze zmiennych lokalnych. Epilog nie będzie dodany, jeśli jest zastosowana instrukcja `„ret 0”`, `„ret Num”`, `„retn”` lub `„retf”`.

Minimalny prolog standardowy to dwie instrukcje:

```
push EBP
mov EBP, ESP
```

Jeśli w definicję podprogramu jest pole `USES`, to asembler dodaje do prologu ładowanie na stos rejestrów zapisanych w tym polu, na przykład jeśli w definicję jest napisane `„USES ESI, EDI”`, to będą dodane instrukcje:

```
push ESI
push EDI
```

Jeśli w podprogramie jest zastosowana dyrektywa `LOCAL`, to do prologu jest dodawana instrukcja, która rezerwuje miejsce na stosie dla lokalnych zmiennych.

Minimalny epilog standardowy zawiera dwie instrukcje:

```
mov ESP, EBP
pop EBP
```

W przypadku stosowania pola `USES` do epilogu są dołączane instrukcje zdejmowania rejestrów opisanych w tym polu, na przykład:

```
push EDI
push ESI
mov ESP, EBP
pop EBP
```

W polu `<parametry_prologu>` mają zastosowanie opcje:

`FORCEFRAME` – dodać prolog i epilog, nawet przy braku argumentów i dyrektywy `LOCAL`,
`LOADDS` – dodać rejestr `DS` do rejestrów zapisywanych w prologu i odczytywanych w epilogu.

Dyrektywa LOCAL

Dyrektywa *LOCAL* rozmieszczona na początku podprogramu rezerwuje miejsce na stosie dla lokalnych zmiennych. Miejsce na stosie jest rezerwowane przez zmniejszenie wartości wskaźnika stosu – rejestru ESP na wielkość równej sumie bajtów wszystkich zmiennych lokalnych.

Na przykład, aby zarezerwować miejsce na stosie dla dwóch zmiennych typu DWORD, należy zastosować dyrektywę:

```
LOCAL zm1:DWORD, zm2:DWORD
```

Po analizie tej przykładowej dyrektywy asembler doda do prologu instrukcję:

```
sub ESP, 8
```

Dyrektywy OPTION PROLOGUE I OPTION EPILOGUE

Dyrektywy *OPTION PROLOGUE* i *OPTION EPILOGUE* sterują generowaniem prologa i epiloga odpowiednio. Dyrektywy mają wpływ na generowanie prologów i epilogów wszystkich podprogramów (funkcji) zdefiniowanych po tych dyrektywach.

Składnia dyrektyw:

```
OPTION PROLOGUE: nazwa_makro | NONE | PROLOGUEDEF
```

```
OPTION EPILOGUE: nazwa_makro | NONE | EPILOGUEDEF
```

Opcja *NONE* tłumy generowanie prologu lub epilogu odpowiednio. Opcja *PROLOGUEDEF* lub *EPILOGUEDEF* przywraca generowanie standardowego prologu lub epilogu odpowiednio.

Opcja *nazwa_makro* umożliwia zamianę standardowego prologu lub epilogu na inny zdefiniowany przez makro ze strukturą:

```
nazwa_makro MACRO procname, flag, parmbytes, \
localbytes, <reglist>, userparms
```

```
...
EXITM %parmbytes
```

```
ENDM
```

Formalne parametry makra:

- *procname* – nazwa podprogramu (funkcji),
- *flag* – 16-bitowy znacznik, w którym zakodowano:
 - w bitach od 0 do 2 typ językowy (000 – nie zdefiniowano, 001 – C, 010 – SYSCALL, 011 – STDCALL, 100 – PASCAL, 101 – FORTRAN, 110 – BASIC)
 - w bicie 4, czy program wywołujący odpowiada za przesunięcie stosu,
 - w bicie 5, czy podprogram jest typu FAR,
 - w bicie 6, czy podprogram jest typu PRIVATE,
 - w bicie 7, czy podprogram jest typu EXPORT,
 - w bicie 8, czy generowanie jest spowodowane przez instrukcję “iret” (“1”), czy – przez instrukcję “ret” (“0”),
- *parmbytes* – ilość bajtów wszystkich argumentów podprogramu,
- *localbytes* – ilość bajtów wszystkich zmiennych lokalnych,
- *reglist* – lista rejestrów w polu *lista_rejestrów* w operatorze *USES*,
- *userpars* – lista parametrów wpisanych do pola *<parametry_prologu>* w definicję podprogramu (funkcji).

Asembler używa tego makra wysyłając do niego parametry faktyczne w kolejności parametrów formalnych.

Aby spowodować wygenerowanie niestandardowego prologu lub epilogu w definicji podprogramu (funkcji) musi być opcja *FORCEFRAME* w polu *<parametry_prologu>*.

Dyrektywy INVOKE I PROTO

Asembler MASM posiada wygodną formę wywołania podprogramu, w której jest zastosowana *dyrektywa INVOKE*:

```
INVOKE wyrażenie_funkcja [[, argument] ...]
```

Korzystając z dyrektywy *INVOKE* programista jest zwolniony od wypisywania instrukcji ładujących na stos argumenty podprogramu. Drugą zaletą jest to, że lista argumentów ma „zwykłą” formę. Przykład:

```
;program wywołujący
INVOKE testProc, param1, param2 ; wywoływanie podprogramu
```

Pole wyrażenie `_funkcja` może zawierać nie tylko nazwę podprogramu (funkcji), a i wyrażenie, którego wynikiem obliczenia jest etykieta.

W łańcuchu argumentów można zamieniać operatory `OFFSET` na operatory `ADDR`, co powoduje lepszą czytelność listy argumentów.

Program stosujący dyrektywę `INVOKE` musi zawierać *dyrektywę* `PROTO`, ponieważ ta dyrektywa informuje asembler o typach argumentów i kolejności przesyłania ich na stos. Dyrektywa `PROTO` ma składnię:

```
_etykieta lub _nazwa_funkcji PROTO [_odległość]  
[_typ_językowy] [[, [_argument] :_typ] ...]
```

Pola `_odległość`, `_typ_językowy`, `_argument` oraz `_typ` mają takie same przeznaczenie jak w składni dyrektywy `PROC`.

Przykład:

```
wsprintfA PROTO C :VARARG  
lstrlenA PROTO :DWORD
```

Stosowanie podprogramów

Instrukcje związane z podprogramami

Z podprogramami są związane instrukcje przedstawione w tab. 6.21 .

Tabela 6.21.

Instrukcje związane z podprogramami

Przeznaczenie	Instrukcja
Wywoływanie podprogramu	<code>call nazwa</code>
Powrót „bliski” (near) z podprogramu. Polega na załadowaniu do licznika rozkazów EIP słowa pobranego ze stosu	<code>ret num</code>
Powrót „daleki” (far) z podprogramu. Polega na załadowaniu do rejestru CS słowa pobranego ze stosu, a do licznika rozkazów EIP następnego słowa pobranego ze stosu	<code>retf num</code>
Wywołanie funkcji o danym numerze, której adres znajduje się w tablicy wektorów przerwań	<code>int num</code>
Warunkowe wywołanie funkcji o numerze 4, gdy ustawiony jest znacznik OF	<code>into</code>
Powrót z procedury przerwania	<code>iret</code>

Ramki stosu

W przypadku przekazywania argumentów do podprogramu przez stos komórki stosu z argumentami tworzą grupy nazywane *ramkami* (*frames*). W zakresie ramki są stosowane dwie kolejności przekazywania argumentów, które często są nazywane konwencjami wywoływania języków C i Pascal.

Konwencja wywoływania języka C określa następującą kolejność umieszczania na stosie:

- ostatni argument z tych, które program wywołujący chce umieścić na stosie,
- przedostatni argument,
- itd.,
- pierwszy z argumentów,
- adres powrotu (typu NEAR lub FAR) (zapisuje procesor) ,
- zawartość EBP (zapisuje podprogram).

W przypadku realizacji konwencji wywoływania języka C jest możliwość operowania ze zmienną liczbą argumentów podprogramu. Ponieważ pierwszy argument zawsze znajduje się w komórce (EBP+8), to po wartości pierwszego argumentu podprogram może ustalić, ile argumentów będzie w tym konkretnym wywołaniu.

Charakterystyczną cechą konwencji wywoływania języka C jest to, że za likwidację ramki (ang. *frame*) odpowiada program wywołujący. Ten program „zna” aktualną liczbę i typy argumentów i po zakończeniu podprogramu zwiększa zawartość rejestru EBP.

Konwencja wywoływania języka PASCAL określa następującą kolejność umieszczania na stosie:

- pierwszy argument,
- następny argument,
- itd.,
- ostatni z argumentów,
- adres powrotu (typu NEAR lub FAR) (zapisuje procesor) ,
- zawartość EBP (zapisuje podprogram).

W przypadku realizacji konwencji wywoływania języka PASCAL liczba argumentów musi być stała.

Charakterystyczną cechą konwencji wywoływania języka PASCAL jest to, że ramkę (ang. *frame*) likwiduje podprogram, dlatego że „zna” liczbę i typy wszystkich argumentów.

Wywołanie podprogramu

Definiując podprogram należy wskazywać typ wywołania: NEAR lub FAR.

Wywołanie podprogramu typu NEAR nazywa się *wywołaniem bliskim*, ponieważ odległość w bajtach między miejscem z instrukcją `call` i początkiem podprogramu jest mniejsza niż rozmiar segmentu i assembler może operować tylko przesunięciem. W tym przypadku przy wykonywaniu rozkazu `call` zapisuje się do stosu tylko zawartość licznika rozkazów EIP, do licznika ładuje się tylko przesunięcie, a zawartość rejestru CS nie zmienia się. Przy powrocie z podprogramu musi być wykonany rozkaz RET, który ładuje do licznika rozkazów to słowo ze stosu, w którym był zapisany stan licznika rozkazów.

Wywołanie podprogramu typu FAR nazywa się *wywołaniem odległym*. W tym przypadku przy wykonywaniu rozkazu `call` są zapisywane na stos zawartości rejestru segmentowego CS i licznika rozkazów EIP,

a do tych rejestrów jest ładowany nowy adres (segment i przesunięcie). Dlatego przy powrocie z podprogramu trzeba wykonać rozkaz RETF, który ładuje wcześniej zapisane wartości do rejestru segmentowego CS i licznika rozkazów EIP.

Parametr „num” instrukcji RET lub RETF to ilość bajtów, na którą należy przesunąć wskaźnik stosu, żeby można było znowu wykorzystać miejsce na stosie, które było użyte do rozmieszczenia lokalnych zmiennych podprogramu.

Wartość zwracana podprogramem jest umieszczana w rejestrze-akumulatorze EAX.

Jeśli program stosuje model „flat”, to wywołanie podprogramów jest bliskie (NEAR), chociaż rozmiar segmentu 4 GB wygląda na „duży”. W podprogramach dla modeli „flat” należy stosować instrukcję RET.

Jawne wywołanie podprogramu

Parametry są przekazywane do podprogramu przez rejestry lub przez stos. Używa się też mieszany wariant przekazywania – część parametrów przez rejestry, a pozostała część przez stos. Jasne, że liczba parametrów przekazywanych przez rejestry jest mocno ograniczona (praktycznie 3, 4 parametry).

Stos można nie używać w ogóle, jeśli przekazywać przez rejestr adres obszaru pamięci z parametrami. Ale mechanizm przekazywania parametrów przez stos jest uniwersalny w takim stopniu, że jest on stosowany nawet przy małej liczbie parametrów.

Typową współpracę programu wywołującego oraz podprogramu w przypadku konwencji wywoływania języka C ilustruje następujący przykład:

```
;;program wywołujący
.MODEL flat, C
...
push param2 ;odkładanie na stos parametru drugiego (4 bajty)
push param1 ;odkładanie parametru pierwszego (4 bajty)
call testProc ; wywoływanie podprogramu
add ESP, 8 ;ominięcie ramki stosu z parametrami
...
;;podprogram
testProc PROC
;prolog standardowy:
push EBP ;przechowywanie EBP na stosie
mov EBP, ESP ;zamiana EBP
...
mov EAX, [EBP+8];korzystanie z parametru pierwszego
...
mov EAX, [EBP+12] ;korzystanie z parametru drugiego
...
mov EAX, wartość_zwrot
;epilog standardowy:
mov ESP, EBP ;zamiana ESP
pop EBP ;przewrócenie EBP ze stosu
ret
testProc ENDP
```

Pierwsze dwie instrukcje podprogramu są prologiem standardowym, a instrukcje „mov ESP, EBP” i „pop EBP” - epilogiem standardowym.

Instrukcja „mov EBP, ESP” daje możliwość stosowania wewnątrz podprogramu wskaźników „[EBP+8]” i „[EBP+12]” do parametrów zapisanych na stosie. W adresie „[EBP+8]” pierwszego z zapisanych argumentów liczba „8” jest sumą 4 bajtów, w których jest zapisany na stosie stan licznika rozkazów EIP, i 4 bajtów, w których jest zapisana poprzednia zawartość rejestru EBP.

W przypadku konwencji języka PASCAL za przesuwanie stosu odpowiada podprogram. W celu przesuwania stosu na ilość bajtów zajmowanych argumentami należy stosować instrukcję „ret ilość_bajtów”.

Wyżej przytoczony przykład w przypadku konwencji wywoływania języka Pascal zmieni się na następujący:

```
;;program wywołujący
.MODEL flat, PASCAL
...
push param1 ;odkładanie parametru pierwszego (4 bajty)
push param2 ;odkładanie na stos parametru drugiego (4 bajty)
call testProc ; wywoływanie podprogramu
```

```

...
;;podprogram
testProc PROC
;prolog standardowy:
push EBP ;przechowywanie EBP na stosie
mov EBP, ESP ;zamiana EBP
...
mov EAX, [EBP+12] ;korzystanie z parametru pierwszego
...
mov EAX, [EBP+8];korzystanie z parametru drugiego
...
mov EAX, wartość_zwrot
;epilog standardowy:
mov ESP, EBP ;zamiana ESP
pop EBP ;przewrócenie EBP ze stosu
ret 8 ;ominięcie ramki stosu z parametrami
testProc ENDP

```

Stosowanie dyrektywy INVOKE

Dyrektywa INVOKE jest lepszą formą wywołania podprogramu w języku MASM, ponieważ programista jest zwolniony od wypisywania instrukcji ładujących na stos argumenty podprogramu, a lista argumentów ma tradycyjną formę. Przykład:

```

;;program wywołujący
INVOKE testProc, param1, param2 ; wywołanie podprogramu

```

Stosując dyrektywę INVOKE należy zapisać na początku programu dyrektywę PROTO, która informuje asembler o typach argumentów.

Przykład:

```

testProc PROTO C :DWORD, :DWORD

```

Zmienne lokalne

Jawna rezerwacja miejsca na stosie

Jeżeli wewnątrz podprogramu potrzebne są zmienne lokalne, to można zarezerwować dla nich miejsce na stosie:

```

;;podprogram
testProc PROC
push EBP ;przechowywanie EBP na stosie
mov EBP, ESP ;zamiana EBP
sub ESP,8 ;rezerwacja miejsca na dwie zmienne typu DWORD
...
mov [EBP-4], EAX ;korzystanie z pierwszej zmiennej lokalnej
...
mov [EBP-8], EAX ;korzystanie z drugiej zmiennej lokalnej
...
mov EAX, wartość_zwrot
mov ESP, EBP ;przewrócenie wartości ESP
pop EBP ;przewrócenie EBP ze stosu
ret
testProc ENDP

```

Stosowanie dyrektywy LOCAL

Metodą niejawną rezerwacji miejsce na stosie dla lokalnych zmiennych jest stosowanie dyrektywy LOCAL na początku podprogramu. Na przykład, aby zarezerwować miejsce na stosie dla dwóch zmiennych typu DWORD, należy zastosować dyrektywę:

```

LOCAL zm1:DWORD, zm2:DWORD

```

Ta przykładowa dyrektywa jest ekwiwalentna jawnej rezerwacji miejsca na stosie:

```

sub ESP, 8

```

Zaletą stosowania dyrektywy LOCAL jest możliwość użycia w podprogramu identyfikatorów zmiennych, na przykład:

```

mov zm1, 15

```



```
shl zml, 4
```

```
*****
```

5.18. Makrodefinicje

Makrodefinicja (makro) jest nazwą ciągu znaków lub instrukcji. W języku MASM do makr są doliczane:

- text macros (makro tekstowe) – nazwy literałów tekstowych,
- macro procedures (makroinstrukcje) – nazwy ciągów instrukcji,
- repeat blocks (dyrektywy powtórzeń) – konstrukcje opisujące cykle,
- macro functions (makrofunkcje) – nazwy funkcji zwracających wartość tekstową,
- predefined macro functions (makrofunkcje zdefiniowane) – nazwy funkcji zdefiniowanych.

Makra mogą być włożone jedno w drugie oraz mogą być rekurencyjne.

5.18.1. Makra tekstowe i dyrektywa **TEXTEQU**

Do definiowania makra tekstowego jest używana *dyrektywa TEXTEQU*, która może występować w trzech wariantach:

```
nazwa      TEXTEQU <literał>
nazwa      TEXTEQU makro_ident | makro_tekst
nazwa      TEXTEQU %wyrażenie
```

Pierwszy wariant dyrektywy **TEXTEQU** jest podobny do dyrektywy **EQU**, może być zamieniony na dyrektywę **EQU** i służy do nadawania imienia literału tekstowemu.

W wariancie drugim ma miejsce nadawanie imienia makrofunkcji („makro_ident”) lub zamiana imienia makra tekstowego („makro_tekst”).

Do przekształcenia wartości wyrażenia w postać tekstową służy trzecia forma dyrektywy **TEXTEQU**.

5.18.2. Makroinstrukcje

W języku MASM makroinstrukcja jest nazwą ciągu instrukcji. Ciąg instrukcji zadeklarowany jako makroinstrukcja jest wstawiany asemblerem w tym miejscu programu, gdzie jest napisana nazwa makra. Tradycyjnie napisanie nazwy makra w programie jest nazywane „wywołaniem makra”.

Makroinstrukcja jest deklarowana za pomocą *dyrektywy MACRO* i ma następującą składnię:

```
_nazwa MACRO [ _parametr[:_kwalifikator]
[, _parametr[:_kwalifikator]]...]
[LOCAL lista_etykiet]
...
ENDM
```

Pole `_kwalifikator` może mieć wartości:

- „REQ” - zaznacza obowiązkowy parametr makroinstrukcji,
- „=<wartość_domyślna>” - ustawia wartość domyślną w przypadku pustego parametru,
- „VARARG” – lista ze zmienną ilością parametrów.

Kwalifikator „VARARG” może być zastosowany tylko dla ostatniego parametru.

Dyrektywa LOCAL wewnątrz makroinstrukcji opisuje zmienną lub etykietę lokalną w stosunku do makroinstrukcji.

W przypadku opisywania etykiety dyrektywa **LOCAL** wskazuje asemblerowi na przystosowanie wartości etykiety do miejsca wywołania makra. Przykład:

```
LOCAL      etyk1, etyk2
```

gdzie `etyk1`, `etyk2` są nazwy etykiet.

Wewnątrz makra można użyć etykiety anonimowej.

Do wywołania makra służy konstrukcja:

```
_nazwa _argument[, _argument]
```

Wewnątrz makroinstrukcji można stosować dyrektywy **IFB**, **IFNB** kompilacji warunkowej, oraz **.ERRB**, **.ERRNB**, **.ERRIDN**, **.ERRIDNI**, **.ERRDIF**, **.ERRDIFN** kontroli programu.

Operatory makroinstrukcji

Wewnątrz makr tekstowych i makroinstrukcji można stosować operatory przedstawione w tab. 5.16. Przykład z operatorami makroinstrukcji:

```
a      EQU      5              ; a = 5
```

```

b    TEXTEQU <10 -a !> 0> ; b = „10 - a > 0”
c    TEXTEQU %10 - a      ; c = „5”

```

Operator zamiany & może być zapisany nie tylko przed parametrem makro, ale i po parametrze. Na przykład następujące dwa makra są równoznaczne:

```

GenKom    MACRO          num
kom&num    DB    „Błąd &num”
ENDM

```

```

GenKom    MACRO          num
kom&num    DB    „Błąd num&”
ENDM

```

Przy wartości 4 parametru num obydwa makra produkują wiersz:

```

kom4 DB    „Błąd 4”

```

Tabela 5.16

Operatory makroinstrukcji

Operator	Nazwa	Opis
&	Substitution Operator	Operator zamiany. Wymusza stosowanie aktualnej wartości parametru makro lub makra tekstowego
<>	Text Delimiters	Nawiasy trójkątne do literału tekstowego
!	Literal-Character Operator	Nawiązuje interpretację występującego po nim znaku jako symbolu
%	Expansion Operator	Operator rozwinięcia. Wymusza rozwinięcie makra tekstowego lub interpretację wyrażenia jako tekstu

Operator rozwinięcia „%” umieszczony na początku wiersza ma zakres działania na cały wiersz. Przykład, w którym operator „%” wymusza rozwinięcie makra tekstowego „lista”:

```

tabl16    LABEL          WORD
lista    TEXTEQU <!!<1,2,3,4!>>
%FOR par, lista
    DW    par, par*10, par*100, par*1000
ENDM

```

Dyrektywy powtórzeń

Dyrektywy powtórzeń REPEAT, WHILE, FOR i FORC są opracowane dla stosowania w makroinstrukcjach, ponieważ asembler przystosowuje wartość niewidocznej etykiety cyklu do miejsca wywołania makra. Jednak dyrektywy powtórzeń REPEAT, WHILE, FOR, FORC można używać nie tylko w makroinstrukcjach.

W wersjach MASM starszych niż 6.1 dyrektywa REPEAT nazywa się REPT, FOR – IRP, a FORC – IRPC. Każda wersja MASM przyjmuje poprzednie nazwy dyrektyw.

Asembler przerabia dyrektywy powtórzeń tylko jeden raz w pierwszym przejściu. Z tego powodu te dyrektywy nie powinny zawierać etykiety, ponieważ wartość etykiet jest ustawiana w drugim przejściu.

Koniec cyklu definiowanego dyrektywą powtórzeń jest oznaczany przez dyrektywę ENDM.

Dyrektywa REPEAT ma parametr _param, który określa ilość powtórzeń:

```

REPEAT    _param
...
ENDM

```

Na przykład za pomocą dyrektywy REPEAT stworzymy tablicę alfabet z literami alfabetu łacińskiego:

```

alfabet    LABEL    BYTE
znak= 'A'
REPEAT    26
    DB    znak
    znak= znak+1
ENDM

```

Dyrektywa WHILE ma parametr _warunek, którego prawdziwość powoduje powtórzenie instrukcji cyklu:

```

WHILE    _warunek
...

```

ENDM

Wewnątrz cyklu można stosować dyrektywę EXITM w celu wcześniejszego wyjścia.

Na przykład stworzymy tablicę alfabet z literami alfabetu łacińskiego tym razem za pomocą dyrektywy

```
WHILE:
alfabet LABEL      BYTE
znak= 'A'
WHILE    znak LE 'Z'
    DB    znak
    znak= znak+1
ENDM
```

Dyrektywy FOR i FORC mają parametr i listę argumentów. Te dyrektywy pozwalają powtórzyć obliczenia dla wartości podanych w liście.

Składnia *dyrektywy FOR* jest następująca:

FOR parametr, <arg1 [,arg2]...>

...;instrukcje cyklu

ENDM

Cykl powtarza się dla każdego argumentu w liście „arg1, arg2, ...”. Na przykład następująco można zdefiniować tablicę z 16 liczbami:

```
tabl16 LABEL      WORD
FOR par, <1,2,3,4>
    DW    par, par*10, par*100, par*1000
ENDM
```

Tekst fragmentu jest ekwiwalentny następującej definicji:

```
tabl16    DW    1, 10, 100, 1000
           DW    2, 20, 200, 2000
           DW    3, 30, 300, 3000
           DW    4, 40, 400, 4000
```

Wewnątrz makra dyrektywa FOR może operować z argumentem typu VARARG, tj. z argumentami w ilości niezdefiniowanej wcześniej.

Na przykład następujące makro generuje tablicę ze zmienną ilością elementów:

```
GenerTabl MACRO      arg:VARARG
    FOR par, <arg>
        DW    par, par*10, par*100
    ENDM
ENDM
```

Wywołanie tego makra z dwoma argumentami:

```
tabl LABEL      WORD
GenerTabl 1,4
```

produkuje tablicę odpowiedniego rozmiaru:

```
tabl    DW    1, 10, 100
        DW    4, 40, 400
```

Parametr dyrektywy FOR może mieć wartość domyślną.

Przykład:

```
GenerTabl MACRO      arg:VARARG
    FOR par:=<1>, <arg>
        DW    par, par*10, par*100
    ENDM
ENDM
```

Wywołanie makra z jednym pustym argumentem:

```
tabl LABEL      WORD
GenerTabl 5,,2
```

produkuje:

```
tabl DW    5, 50, 500
      DW    1, 10, 100
      DW    2, 20, 200
```

Dyrektywa FORC ma składnię podobną do dyrektywy FOR:

IRPC parametr, <ciag_znaków>

```
...;ciało cyklu
```

```
ENDM
```

Przy wywoływaniu tej dyrektywy cykl powtarza się dla każdego znaku w ciągu znaków.

Na przykład następująco można zdefiniować 3 zmiennych:

```
FORC par, <ACE>
```

```
zm_&par DB '&par'
```

```
ENDM
```

Kompilator wyprodukuje 3 wierszy:

```
zm_A DB 'A'
```

```
zm_C DB 'C'
```

```
zm_E DB 'E'
```

Makrofunkcje

Makrofunkcją w języku MASM jest nazywana makroinstrukcja, która zwraca wartość za pomocą dyrektywy EXITM z parametrem tekstowym:

```
EXITM _param_tekstowy
```

Parametr tekstowy może być literałem w nawiasach kątowych, na przykład <tak>, <-1>, lub zawierać operator „%”, na przykład %nn.

Przykład 5.1. Makrofunkcja do obliczania silni

Silnia MACRO nn:REQ

```
LOCAL s,p
```

```
p = nn
```

```
s = 1
```

```
WHILE p GT 1
```

```
s = s * p
```

```
p = p - 1
```

```
ENDM
```

```
EXITM %s
```

```
ENDM
```

Wywołanie makrofunkcji z przykładu 5.1:

```
silnDD Silnia (10) ; zmienna "siln" ma wartość 10!
```

Makrofunkcje predefiniowane

Do makrofunkcji predefiniowanych należą:

@SizeStr – zwraca rozmiar wierszu (tablicy tekstowej),

@CatStr – łączy dwa lub więcej wierszy w jeden wiersz,

@SubStr – kopiuje podwiersz zadanego rozmiaru od pozycji startowej w wierszu bazowym,

@InStr – wyszukuje zadany podwiersz od pozycji startowej w wierszu bazowym i zwraca pozycję pierwszego znaku.

Przytoczone makrofunkcje bazują na dyrektywach SIZESTR, CATSTR, SUBSTR, INSTR odpowiednio.

Makrofunkcje różnią się od odpowiednich dyrektyw tym, że umożliwiają stosowanie nawiasów. Przykład:

```
liczb = 6
```

```
w1 TEXTEQU @CatStr(<Wynik = >, %liczb) ; "Wynik = 6"
```

```
roz= @SizeStr(w1) ; rozm=9
```

Składnie makrofunkcji:

```
@SizeStr(wiersz)
```

```
@CatStr(wiersz [[, wiersz]...])
```

```
@SubStr(wiersz_bazowy, pozycja_startowa [, rozmiar])
```

```
@InStr ([pozycja_startowa], wiersz_bazowy, podwiersz)
```

Stosowanie makrodefinicji

Makrodefinicja jest nazwą symboliczną fragmentu danych lub kodu. W miejscu wywołania makrodefinicji kompilator wpisuje kod odpowiadający makrodefinicji.

Makro wprowadzamy w przypadku ukazania się w programie fragmentów z jednakową strukturą. Analizując wystąpienia takiego fragmentu można wydzielić parametry makra, które różnią się tylko wartościami w różnych fragmentach.

Na przykład programy konsolowe zawierają fragmenty związane z otrzymaniem deskryptorów wejściowego i wyjściowego buforów konsoli za pomocą funkcji GetStdHandle:

- fragment 1:

```
push  STD_OUTPUT_HANDLE
call  GetStdHandle
mov  hout,EAX ; deskryptor wyjściowego bufora konsoli
```

- fragment 2:

```
push  STD_INPUT_HANDLE
call  GetStdHandle
mov  hinp,EAX ; deskryptor wejściowego bufora konsoli
```

Widoczna strukturalna jednakowość fragmentów sprowadza do makra, na przykład z nazwą **PODAJDESKRKONSOLI** i dwoma parametrami formalnymi:

```
PODAJDESKRKONSOLI MACRO handle, deskrypt
push  handle
call  GetStdHandle
mov  deskrypt,EAX ;; deskryptor bufora konsoli
ENDM
```

W wywołaniach tego makra podajemy parametry faktyczne:

```
PODAJDESKRKONSOLI  STD_OUTPUT_HANDLE, hout
```

oraz

```
PODAJDESKRKONSOLI  STD_INPUT_HANDLE, hinp
```