ROZDZIAŁ ÓSMY : **MASM: DYREKTYWY I PSEUDO-OPCODY**

Instrukcje takie jak mov ax,0 i add ax, bx są niezrozumiałe dla mikroprocesora. W takiej postaci w jakiej te instrukcje się pojawiają, są one jeszcze czytelnymi dla ludzi postaciami instrukcji 80x86.80x86 reaguje na polecenia takie jak B80000 i 03C3.Assembler jest programem który konwertuje łańcuchy takie jak mov ax, 0 do kodu maszynowego 80x86 „B80000".Każdy program w języku asemblera zawiera instrukcje takie jak mov ax, 0.Assembler konwertuje każdy asemblerowy plik źródłowy do kodu maszynowego – binarny odpowiednik programu asemblerowego. Pod tym względem program asemblerowy jest jak kompilator, odczytuje plik źródłowy ASCII z dysku i tworzy na wyjściu program języka maszynowego. Główna różnica między kompilatorem dla języka wysokiego poziomu (HLL) takim jak Pascal i asemblerem jest taka, że kompilator zwykle emituje kilka instrukcji maszynowych dla każdej instrukcji pascalowskiej .Asembler generalnie emituje pojedynczą instrukcję maszynową dla każdej instrukcji języka asemblera.

Próba napisania programu w języku maszynowym (tj. w binarnym) nie jest szczególnie miła. Proces ten jest bardzo nużący, skłonny do błędów i nie proponujący żadnych korzyści przy programowaniu w języku asemblera .Jedyną główną wadą języka asemblera przy czystym kodzie maszynowym jest to ,że musimy najpierw assemblować i linkować program przed jego wykonaniem. Jednakże, próba asemblowania kodu ręcznie będzie trwała dłużej niż mała ilość czasu jaką poświęci asembler na wykonanie tego za nas.

Jest inna wada nauki języka asemblera .Asembler taki jak Microsoft's Macro Assembler (MASM) dostarcza dużej liczby możliwości dla programisty asemblerowego. Chociaż nauka o tych możliwościach zabiera sporo czasu, są one bardzo użyteczne i warte włożonego wysiłku.

8.0 WSTĘP

Podobnie jak w Rozdziale Szóstym, dużo informacji w tym rozdziale jest materiałami odnośnymi. Podobnie jak każda sekcja odnośna, jakaś wiedza jest niezbędna, inny materiał jest przydatny ale opcjonalny, a niektórych materiałów możemy nigdy nie używać podczas pisania programów. Poniższa lista przedstawia informacje w tym tekście. Symbol „•" oznacza materia niezbędny, symbol „⊗" oznacza opcjonalny i mniej użyteczny temat.

• Format źródłowy instrukcji języka asemblera
⊗ Licznik lokacji
• Symbole i identyfikatory
• Stałe
• Deklaracje procedur
⊗ Segmenty w programie języka asemblera
• Zmienne
• Typy symboli
• Wyrażenia adresowe (późniejsze podsekcje zawierają materiał zaawansowany)
⊗ Warunkowe asemblowanie
⊗ Makra
⊗ Dyrektywy listowania
⊗ Oddzielne asemblowanie

8.1 INSTRUKCJE JĘZYKA ASSEMBLERA

Instrukcje w języku assemblera w pliku źródłowym używają następującego formatu:

{Etykieta} {Mnemonik {Operand}} {;Komentarz}

Każda powyższa jednostka jest polem Cztery powyższe pola są to pole etyliety, pole mnemonika, pole operandu i pole komentarza.

Pole etykiety jest (zazwyczaj) polem opcjonalnym zawierającym etykietę symboliczną dla bieżącej instrukcji. etykiety są używane w języku assemblera, podobnie jak w HLL, do oznaczania linii jako celu skoków GOTO .Możemy również wyszczególnić nazwę zmiennej, nazwę procedury i inne jednostki używające etykiet symbolicznych. Przez większość czasu pole etykiety jest opcjonalne, w znaczeniu, że etykieta może być obecna tylko, jeśli chcemy etykietę na tej szczególnej linii. Niektóre mnemoniki, jednak wymagają etykiety, inne nie. Generalnie ,powinniśmy zawsze zaczynać nasze etykiety w pierwszej kolumnie (uczyni to nasz program łatwiejszym w czytaniu).

Mnemonik jest nazwą instrukcji (np. mov, add itp.)Słowo mnemonik oznacza wspomaganie pamięci. mov jest dużo łatwiejsze do zapamiętania niż binarny odpowiednik instrukcji mov! Nawiasy klamrowe oznaczają, że ta pozycja jest opcjonalna. Zauważmy jednak, że nie możemy mieć operandu bez mnemonika.

Pole mnemonika zawiera instrukcję assemblera .Instrukcje są dzielone na trzy klasy :instrukcje maszynowe 80x86,dyrektywy assemblera i pseudo opcody .Instrukcje 80x86 są oczywiście mnemonikami assemblera ,które odpowiadają rzeczywistym instrukcjom 80x86 wprowadzonym w Rozdziale Szóstym.

Dyrektywy assemblera są instrukcjami specjalnymi które dostarczają informacji do assemblera ale nie generują żadnego kodu. Przykłady obejmują dyrektywę segment, equ, assume i end. Te mnemoniki nie są ważnymi instrukcjami 80x86.Są one tylko informacjami dla assemblera ,niczym więcej.

Pseudo-opcody są wiadomością dla assemblera, podobnie jak dyrektywy, jednak pseudo-opcod wyemituje bajt kodu wynikowego.. Przykłady pseudo-opcodów obejmują byte ,word, dword, qword i tbyte. Instrukcje te emitują bajty danych wyszczególnione przez ich operandy ale nie są prawdziwymi instrukcjami maszynowymi 80x86.

Pole operandu zawiera operandy lub parametry ,dla instrukcji wyszczególnionej w polu mnemonika. Operandy nigdy nie pojawiają się w wierszu same z siebie. Typ i liczba operandów (zero, jeden, dwa lub więcej) zależy wyłącznie od określonej instrukcji.

Pole komentarza pozwala nam opisać każdą linie kodu źródłowego w naszym programie. .Zauważmy ,że pole komentarza zawsze zaczyna się od średnika. Kiedy asembler przetwarza linię tekstu, kompletnie ignoruje wszystko w linii źródłowej występujące za średnikiem.

Każda instrukcja języka assemblera pojawia się we własnym wierszu w pliku źródłowym. nie możemy mieć wielu instrukcji w pojedynczej linii. Z drugiej strony, ponieważ wszystkie pola w instrukcji języka assemblera są opcjonalne, linie puste są w porządku. Możemy użyć pustych linii gdziekolwiek w naszym pliku źródłowym. Linie puste są użyteczne przy rozmieszczaniu pewnych sekcji kodu, czyniąc je łatwiejszymi do czytania.

Microsoft Macro Assembler jest asemblerem o swobodnej formie. Różne pola instrukcji języka assemblera mogą pojawiać się w każdej kolumnie (chociaż lepiej ,żeby pojawiały się we właściwym porządku) Każda liczba spacji lub tabulatorów może oddzielać różne pola w instrukcji. Dla assemblera, te dwie sekwencje kodu są identyczne

```
-------------------------------------------------------
          mov     ax, 0
          mov     bx, ax
          add     ax ,dx
          mov     cx, ax
-------------------------------------------------------
mov               ax,                  0
          mov bx,                      ax
     add                 ax, dx
                         mov           cx,ax
-------------------------------------------------------
```

Pierwsza sekwencja kodu jest dużo łatwiejsza do odczytu niż druga (jeśli tak nie sądzisz, być może powinieneś zobaczyć się z lekarzem!)

Umieszczenie etykiety w kolumnie jeden, mnemoników w kolumnie 17 (dwa tabulatory),pola operandu w kolumnie 25 (trzy tabulatory) i komentarza około kolumny 41 lub 49 (pięć lub sześć tabulatorów) tworzy najlepiej wyglądający listing. Programy w języku assemblera, takie jak ten , są dosyć trudne do odczytania. formatowanie naszego listingu pomoże uczynić go łatwiejszym do odczytu i uczyni łatwiejszym do pielęgnacji.

Możemy mieć sam komentarz w linii. W takim przypadku, umieszczamy średnik w kolumnie jeden i używać całej linii dla komentarza ,przykłady:
;Następująca sekcja kodu pozycjonuje kursor w górnej, lewej części ekranu:

```
            mov     X, 0
            mov     Y, 0
```

8.2 LICZNIK LOKACJI

Przypomnijmy, że wszystkie adresy w przestrzeni pamięci 80x86 składa się z adresu segmentowego i offsetu wewnątrz segmentu. Assembler, w trakcie konwertowania naszego pliku źródłowego do kodu wynikowego, musi zachować ścieżkę offsetu wewnątrz bieżącego segmentu. Licznik lokacji jest zmienną assemblera która to obsługuje.

Kiedy tworzymy segment w naszym pliku źródłowego języka assemblera, assembler kojarzy z nim wartość bieżącego licznika lokacji. Licznik lokacji zawiera bieżący offset do tego segmentu. Pierwotnie (kiedy assembler po raz pierwszy napotyka segment) licznik lokacji jest ustawiony na zero. Kiedy napotyka instrukcję lub pseudo-opcodu, MASM zwiększa licznik lokacji dla każdego bajtu zapisanego w pliku kodu wynikowego. Na przykład, MASM zwiększa licznik lokacji o dwa po napotkaniu mov ax, bx ponieważ instrukcja ta jest dwubajtowa.

Wartość licznika lokacji zmienia się w całym procesie asemblacji. Zmienia się dla każdej linii kodu w naszym programie, kiedy tworzymy kod wynikowy .Będziemy używać terminu licznik lokacji w znaczeniu wartości licznika lokacji przy poszczególnej instrukcji przed wygenerowaniem jakiegoś kodu. Rozważmy następujące instrukcje języka assemblera:

```
0:          or      ah, 9
3:          and     ah,0c9h
6:          xor     ah,40
9:          pop     cx
A:          mov     al.,cl
C:          pop     bp
D:          pop     cx
E:          pop     dx
F:          pop     ds.
10:         ret
```

Instrukcje or, and i xor ,wszystkie są długości trzech bajtów, instrukcja mov jest dwubajtowa; pozostałe instrukcje wszystkie są jednobajtowe. Jeśli te instrukcje pojawią się na początku segmentu, licznik lokacji będzie taki sam jak liczby które pojawiają się bezpośrednio na lewo od każdej powyższej instrukcji. Na przykład, instrukcja or zaczyna się od offsetu zero, ponieważ instrukcja or jest trzybajtowa ,następna instrukcja (and) wystąpi pod offsetem trzy. Podobnie and jest trzybajtowa ,więc xor wystąpi od offsetu sześć itd.

8.3 SYMBOLE

Rozważmy na chwilę instrukcję jmp. Instrukcja ta przyjmuje formę:

```
            jmp     cel
```

Cel jest adresem przeznaczenia. Wyobraźmy sobie jak żmudne byłoby gdybyśmy musieli w rzeczywistości wyszczególnić adres docelowy pamięci jako wartość liczbową. Jeśli kiedyś programowaliśmy w BAISICu, doświadczylibyśmy około 10% problemów jakie mielibyśmy w języku assemblera gdybyśmy musieli wyszczególniać cel dla jmp poprzez adres.

Dla ilustracji, przypuśćmy, że chcielibyśmy skoczyć do pewnej grupy instrukcji którą jeszcze piszemy.. Jaki jest adres instrukcji docelowej? Jak możemy powiedzieć że napisaliśmy jakąś instrukcję przed instrukcją docelową? Co się zdarzy jeśli zmienimy program (pamiętajmy, wprowadzenie i kasowanie instrukcji powoduje, że wartość licznika lokacji dla wszystkich następnych instrukcji wewnątrz tego segmentu zmienia się).Na szczęście ,wszystkie te problemy są zmartwieniem programistów języka maszynowego. Programiści języka assemblera mogą zająć się adresami w dużo bardziej rozsądny sposób – poprzez użycie adresów symbolicznych.

Symbol, identyfikator lub etykieta, jest nazwą powiązana z jakąś szczególną wartością. Wartość ta może być offsetem wewnątrz segmentu, stałą, łańcuchem, adresem segmentowym, offsetem wewnątrz rekordu lub nawet operandem dla instrukcji. W każdym razie, etykieta zapewnia nam możliwości do przedstawiania niezrozumiałych wartości ,jako dobrze znanej mnemonicznej nazwy.

Nazwa symboliczna składa się z sekwencji liter, cyfr i znaków specjalnych, z następującymi ograniczeniami:

- Symbol nie może zaczynać się od cyfry

- Nazwa może mieć każdą kombinację dużych i małych liter. Assembler traktuje duże i małe litery równorzędnie.
- Symbol może zawierać każdą liczbę znaków, jednak tylko pierwsze 31 jest używanych. Assembler ignoruje wszystkie znaki po trzydziestym pierwszym
- Symbole „_,$,? I @” mogą się pojawiać gdziekolwiek wewnątrz symbolu. jednak ,$ i ? są symbolami specjalnymi; nie możemy tworzyć symbolu składającego się wyłącznie z tych dwóch symboli.
- Symbol nie może pokrywać się z żadną z nazw ,które są zarezerwowanymi symbolami. Następujące symbole są zarezerwowane:

%out	.186	.286	.286P
.287	.386	.386P	.387
.486	.486P	.8086	.8087
.ALPHA	.BREAK	.CODE	.CONST
.CREF	.DATA	.DATA?	.DOSSEG
.ELSE	.ELSEIF	.ENDIF	.ENDW
.ERR	.ERR1	.ERR2	.ERRB
.ERRDEF	.ERRDIF	.ERRDIFI	.ERRE
.ERRIDN	.ERRIDNI	.ERRNB	.ERRNDEF
.ERRNZ	.EXIT	.FARDATA	.FARDATA?
.IF	.LALL	.LFCOND	.LIST
.LISTALL	.LISTIF	.LISTMACRO	.LISTMACROALL
.MODEL	.MSFLOAT	.NO87	.NOCREF
.NOLIST	.NOLISTIF	.NOLISTMACRO	.RADIX
.REPEAT	.UNTIL	.SALL	.SEQ
.SFCOND	.STACK	.STARTUP	.TFCOND
.UNTIL	.UNTILCXZ	.WHILE	.XALL
.XCREF	.XLIST	ALIGN	ASSUME
BYTE	CATSTR	COMM	COMMENT
DB	DD	DF	DOSSEG
DQ	DT	DW	DWORD
ECHO	ELSE	ELSEIF	ELSEIF1
ELSEIF2	ELSEIFB	ELSEIFDEF	ELSEIFDEF
ELSEIFE	ELSEIFIDN	ELSEIFNB	ELSEIFNDEF

END	ENDIF	ENDM	ENDP
ENDS	EQU	EVEN	EXITM
EXTERN	EXTRN	EXTERNDEF	FOR
FORC	FWORD	GOTO	GROUP
IF	IF1	IF2	IFB
IFDEF	IFDIF	IFDIFI	IFE
IFIDN	IFIDNI	IFNB	IFNDEF
INCLUDE	INCLUDELIB	INSTR	INVOKE
IRP	IRPC	LABEL	LOCAL
MACRO	NAME	OPTION	ORG
PAGE	POPCONTEXT	PROC	PROTO
PUBLIC	PURGE	PUSHCONTEXT	QWORD
REAL4	REAL8	REAL10	RECORD
REPEAT	REPT	SBYTE	SDWORD
SEGMENT	SIZESTR	STRUC	STRUCT
SUBSTR	SUBTITLE	SUBTTL	SWORD
TBYTE	TEXTEQU	TITLE	TYPEDEF
UNION	WHILE	WORD	

W dodatku, wszystkie poprawne nazwy instrukcji 80x86 i nazwy rejestrów są również zarezerwowane. Zauważmy ,że lista ta stosuje się dla MASMa w wersji 6.0.Wcześnejsze wersje tego assemblera mają mniej zarezerwowanych słów. Późniejsze wersje mogą mieć więcej

Kilka przykładów poprawnych symboli:

L1	Bletch	RightHere
Right_Here	Item1	_Special
$1234	@Home	$_@1
Dollar$	WhereAmI	@1234

$1234 i @1234 są zupełnie poprawne, chociaż mogą wydawać się dziwne.

Kilka przykładów niewłaściwych symboli:

1TooMany - zaczyna się cyfrą

Hello.There -zawiera kropkę w środku symbolu
$ -nie może być samodzielnego znaku $ i >
LABEL -zarezerwowane słowo assemblera
Right Here -Symbol nie może zawierać spacji
Hi,There - lub innego znaku specjalnego poza _,?,$ i @

Symbolom, jak wspomniano wcześniej, można przydzielić wartości liczbowe (takie jak wartości licznika lokacji),łańcuchy lub nawet całe operandy. Wyjaśnijmy sobie jedną rzeczy, assembler przydziela typ do każdego symbolu. Przykłady typów near, far, byte, word, double word, quad word, text i string. jak zadeklarować etykiety pewnych typów jest tematem reszty tego rozdziału. Zauważmy, że assembler zawsze przydziela jakiś typ do etykiety i będzie dozorował czy próbujemy użyć etykiety w miejscu gdzie nie jest dozwolony taki typ etykiety.

8.4 STAŁE ZNAKOWE

MASM jest zdolny do przetwarzania pięciu różnych typów stałych: całkowite, całkowite upakowane dziesiętne kodowane binarnie, liczby rzeczywiste, łańcuchy i tekst. W rozdziale tym zajmiemy się tylko stałymi całkowitymi, rzeczywistymi, łańcuchami i tekstem. Po więcej informacji o wartościach całkowitych upakowanych BCD, proszę zgłosić się do Przewodnika MASM.

Stała znakowa jest jedną której wartość jest ukryta pod znakami, które stanowią stałą. Przykłady stałych znakowych:

- 123
- 3.14159
- „Łańcuchowa Stała Znakowa"
- 0FABCh
- ‘A’
- <Stała Tekstowa>

Oprócz ostatniego przykładu, większość stałych znakowych powinny być dobrze znane każdemu kto pisał programy w językach takich jak Pascal lub C++. Stałe tekstowa są specjalnymi formami łańcuchów które pozwalają na zastąpienie tekstowe podczas asemblacji.

Przedstawianie stałych znakowych odpowiada temu co normalnie moglibyśmy oczekiwać dla „wartości rzeczywistego słowa" .Stałe znakowe są również znane jako „stałe nie symboliczne" ponieważ używają wartości rzeczywistych ,zamiast jakichś nazw symbolicznych, wewnątrz naszego programu. MASM również pozwala nam zdefiniować symboliczną lub jawną stałą w programie, ale więcej o tym później.

8.4.1 STAŁE CAŁKOWITE

Stała całkowita jest wartością liczbową, która może być określona jako binarna, dziesiętna lub heksadecymalna. Wybór bazy (lub podstawy systemu liczbowego) należy do nas. Poniższa tablica pokazuje poprawne cyfry dla każdej podstawy:

Name	Base	Valid Digits
Binary	2	0 1
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Tablica 35 Cyfry używane dla każdej podstawy systemu liczbowego

Dla odróżnienia pomiędzy liczbami w kilku systemach liczbowych, używamy znaku przyrostka. Jeśli, kończymy liczbę „b" lub „B", wtedy MASM zakłada, że jest to liczba binarna. Jeśli zawiera każdą inną cyfrę niż zero lub jeden assembler wygeneruje błąd .Jeśli przyrostek to „t", "T" ,"d" lub „D", wtedy assembler zakłada, że jest to wartość dziesiętna (o podstawie 10).Przyrostek „h" lub „H" wybierze podstawę heksadecymalną.

Wszystkie stałe całkowite muszą zaczynać się cyfrą dziesiętną, wliczając w to stałe heksadecymalne. Przedstawienie wartości „FDED" musi wyszczególnić 0FDEDh.Czołowa cyfra dziesiętna jest wymagana przez assembler ,żeby mógł rozróżniać pomiędzy symbolami i stałymi numerycznymi; pamiętajmy ,"FDED" jest zupełnie poprawnym symbolem MASMa.
Przykłady:

0F000h 12345d 0110010100b

1234h 100h 08h

Jeśli nie wyszczególnimy przyrostka po stałej liczbowej, assembler użyje bieżącej, domyślnej podstawy .Domyślną podstawą jest podstawa dziesiętna. dlatego też, możemy zazwyczaj wyszczególnić wartości dziesiętne bez dokładania znaku „D”. Dyrektywa assemblera radix może być użyta do zmiany domyślnej podstawy systemu liczbowego na inną bazę .instrukcja .radix przyjmuje następującą formę:

```
.radix    base            ;opcjonalnie komentarz
```

Base jest domyślną wartością między 2 a 16.

Instrukcja .radix skutkuje jak tylko MASM napotka ją w pliku źródłowym .Wszystkie instrukcje przed instrukcją .radix będą używały poprzedniej domyślnej podstawy dla stałej liczbowej .Poprzez odrobinę wielokrotności instrukcji .radix w całym naszym pliku źródłowym, możemy przełączać domyślną podstawę pomiędzy kilkoma wartościami zależnie od tego jaki jest najbardziej dogodny w danym punkcie programu.

Generalnie, dziesiętna podstawa jest dobra jako podstawa domyślna, więc instrukcja .radix nie używa jej często .Jednakże, w obliczu wejścia do gigantycznej tablicy wartości heksadecymalnych, możemy zachować wiele typów poprzez czasowe przełączenie do podstawy 16 przed tablicą i przełączenia z powrotem do 10 po tablicy .Notka: jeśli domyślna podstawa jest heksadecymalna, powinniśmy użyć przyrostka „I” do oznaczenia wartości dziesiętnej ponieważ MASM może pomylić przyrostek „D” z cyfrą heksadecymalną.

8.4.2 STAŁE ŁAŃCUCHOWE

Stała łańcuchowa jest sekwencją znaków otoczonych przez apostrofy lub cudzysłowy.

Przykłady:

„to jest łańcuch”

‘Więc i to’

Możemy swobodnie umieszczać apostrofy wewnątrz stałej łańcuchowej otoczonej cudzysłowem i vice versa. Jeśli chcemy umieścić apostrof wewnątrz łańcucha ograniczonego przez apostrofy, musimy umieścić następną parę apostrofów w łańcuchu. Np.

‘Co się ‘’ stało?’

Cudzysłów pojawia się wewnątrz łańcucha ograniczonego przez cudzysłów muszą również zdublowane np.

„Microsoft twierdzi „”Nasz software jest bardzo szybki.”” Czy im wierzysz?”

Chociaż możemy zdublować apostrofy i cudzysłowy jak pokazano w powyższych przykładach, łatwiejszym sposobem zawarcia tych znaków w łańcuchu jest użycie innego znaku jako ogranicznika łańcucha.:

„Co się stało?”

‘Microsoft twierdzi „Nasz software jest bardzo szybki.” Czy im wierzysz?’

Jedyny raz kiedy zdublowanie cudzysłowu lub apostrofu jest absolutnie konieczne w łańcuchu jest to, kiedy łańcuch zawiera oba symbole. Rzadko się to zdarza w rzeczywistych programach.

Podobnie jak języki programowania C i C++, istnieje subtelna różnica pomiędzy wartością znaku a wartością łańcucha. Pojedynczy znak (to znaczy, łańcuch o długości jeden) może pojawiać się gdziekolwiek MASM pozwala na stałe całkowite lub łańcuch. jeśli wyszczególnimy stałą znakową tam gdzie MASM spodziewa się stałej całkowitej, MASM użyje kodu ASCII tego znaku jako wartości całkowitej .Łańcuchy (których długość jest większa niż jeden) są dozwolone tylko wewnątrz pewnych kontekstów

---

8.4.3 STAŁE RZECZYWISTE

Wewnątrz pewnych kontekstów możemy użyć stałych zmienno przecinkowych .MASM pozwala nam wyrazić stałą zmiennoprzecinkową na jedną z dwóch postaci :notacji dziesiętnej lub naukowej. Formy te są całkiem podobne dla formatu dla liczb rzeczywistych używanych przez Pascala, C i inne HLL’e .

Forma dziesiętna jest sekwencją cyfr zawierającą punkt dziesiętny w odpowiedniej pozycji liczby:

1.0 3.14159 625,25 -128.0 0.5

Notacja naukowa jest również identyczna do form używanych przez różne HLL’e:

1e5 1.576e-2 -6.02e-10 5.34e+12

Dokładny zakres precyzji liczb zależy od naszego pakietu zmiennoprzecinkowego .Jednakże MASM generalnie emituje dane binarne dla powyższych stałych które są kompatybilne z koprocesorem arytmetycznym 80x87.Forma ta odpowiada formatowi liczbowemu wyspecyfikowanemu przez standard IEEE dla wartości zmienno przecinkowych .W szczególności stała 1.0 nie jest binarnym odpowiednikiem jedynki całkowitej.

8.4.4 STAŁE TEKSTOWE

Stałe tekstowe nie są tym samy co stałe łańcuchowe. Stała tekstowa jest zastępowana dosłownie podczas procesu asemblacji. Na przykład, znaki 5[bx] mogą być stałą tekstową powiązaną z symbolem VAR1.Podczas asemblacji, instrukcja w postaci mov ax,VAR1,będzie skonwertowana do instrukcji mov ax, 5[bx].

Tekstowe porównania są całkiem użyteczne w MASM ponieważ MASM często upiera się przy długich łańcuchach tekstu dla prostego operandu języka assemblera. Używając tekstowych porównań pozwala nam na upraszczanie takich operandów przez zastąpienie łańcucha tekstu przez pojedynczy identyfikator w instrukcji.

Stała tekstowa skalda się z sekwencji znaków otoczonych przez symbole „<” i „>”.Na przykład stała tekstowa 5[bx] mogłaby być normalnie zapisana jako <5[bx]>.kiedy wystąpi zastąpienie tekstu, MASM usunie znaki ograniczające „<” i „”>”.

8.5 DEKLAROWANIE STAŁYCH UŻYWAJĄCYCH DYREKTYW RÓWNOŚCI

Stała jawna jest nazwą symbolu która przedstawia jakąś stałą wartość podczas procesu asemblacji. To znaczy ,jest to nazwa symboliczna która przedstawia jakąś wartość. Dyrektywy równości to mechanizm MASM używany do deklaracji stałych symbolicznych. Dyrektywy równości przybierają trzy podstawowe formy:

```
symbol      equu        wyrażenie
 symbol     =           wyrażenie
symbol      tekstequ    wyrażenie
```

Operand wyrażenia jest wyrażeniem liczbowym lub łańcuchem tekstu. Symbol jest daną wartością i typem wyrażenia. Dyrektywy equ i „=” były w MASM, ponieważ początkowo Microsoft dodał dyrektywę textequ do MASM 6.0

Celem dyrektywy „=” jest zdefiniowanie symboli, które mają całkowitą (lub pojedynczy znak) wartość z nimi związaną. Dyrektywy te nie zezwalają na operandy rzeczywiste, łańcuchowe lub tekstowe. Jest to podstawowa dyrektywa jaką powinniśmy używać do tworzenia stałych liczbowych symbolicznych w naszych programach .Kilka przykładów:

```
NumElements     =       16
                .
                .
                .
Array           byte    NumElements  dup (?)
                .
                .
                .
                mov     cx, NumElements
                mov     bx, 0
ClrLoop:        mov     Array[bx],0
                inc     bx
                loop    ClrLoop
```

Dyrektywa textequ definiuje symbol zastępujący tekst. Wyrażenie w polu operandu musi być stałą tekstową ograniczoną między symbolami „<” i „>”.Ilekroć MASM napotka te symbole wewnątrz instrukcji, zastąpi tekst w polu operandu dla tego symbolu. Programiści zazwyczaj używają tego dyrektywy równości do zachowania typu lub uczynienia jakiegoś kodu bardziej czytelnym:

```
Count       textequ <6[bp]>
DataPtr     textequ <8[bp]>
            -
            -
            -
            les     bx, DataPtr         ;to samo co les bx, 8[bp]
            mov     cx, Count           ;to samo co mov cx, 6[bp]
            mov     al., 0
ClrLp:      mov     es:[bx], al
            inc     bx
            loop    ClrLp
```

Zauważmy, że jest to zupełnie poprawna dyrektywa typu - symbol z pustym operandem używającym dyrektywy równości jak następuje:

```
BlankEqu    textequ     <>
```

Celem takiej dyrektywy równości będzie czyszczenie w segmencie asemblacji warunkowej i makrach.

Dyrektywa equ dostarcza prawie nadzbioru zdolności dyrektyw „=” i textequ. Pozwala na operandy, które są numeryczne ,tekstowe lub stałymi literałami łańcuchowymi. Poniżej pokazano poprawne zastosowanie dyrektywy equ:

```
One         equ     1
Minus1      equ     -1
```

```
TryAgain      equ     'Y'
StringEqu     equ     „hello there”
TxtEqu        equ     <4[si]>
              -
              -
              -
HTString      byte    StrinEqu       ;to samo co HTString equ „hello there”
              -
              -
              -
              mov     ax, TxtEqu     ;to samo co mov ax, 4[si]
              -
              -
              -
              mov     bl, One        ;to samo co mov bl, 1
              cmp     al.,TryAgain   ;to samo co cmp al., 'Y'
```

Stałe jawne zadeklarowane z dyrektywami równości pomogą nam sparametryzować program. Jeśli użyjemy tych samych wartości, łańcuchów lub tekstu wiele razy wewnątrz programu, używając symbolicznych dyrektyw równości uczynimy go dużo łatwiejszym na zmiany tych wartości w przyszłości modyfikując ten program .Rozważmy następujący przykład:

```
Array         byte    16 dup (?)
              -
              -
              -
              mov     cx, 16
              mov     bx,0
ClrLoop:      mov     Array[bx],0
              inc     bx
              loop    ClrLoop
```

Jeśli zdecydujemy, że chcemy mieć Array 32 elementowy zamiast 16,będziemy musieli poszukać w całym naszym programie i zlokalizować każde odniesienie do tej danej i zmodyfikować odpowiednio stałą znakową. Wtedy istnieje możliwość, że nie zmodyfikujemy jakiejś sekcji kodu, wprowadzając błąd do naszego programu. Z drugiej strony, jeśli użyjemy stałej symbolicznej NumElements pokazanej wcześniej, będziemy musieli tylko zmienić pojedynczą instrukcję w naszym programie ,zreassemblować go, i jesteśmy w domu; MASM automatycznie uaktualni wszystkie odnośniki używające NumElements.

MASM pozwala nam przedefiniować symbole zadeklarowane dyrektywą „=”.To znaczy, że następujące przykłady są poprawne:

```
JakiśSymbol       =       0
                  -
                  -
                  -
JakiśSymbol       =       1
```

Ponieważ możemy zmieniać wartości stałych w programie ,zasięg symbolu (gdzie symbol ma szczególną wartość) staje się ważny. Jeśli nie moglibyśmy redefiniować symbolu, moglibyśmy się spodziewać, że symbol ma tą stałą wartość wszędzie w programie. Zważywszy, że możemy redefiniować stałą, zasięg symbolu nie może obejmować całego programu. MASM używa jednego rozwiązania, zasięg stałej jawnej jest od punktu gdzie jest ona zdefiniowana do punktu gdzie jest ona redefiniowana. Ma to jedną ważna konsekwencję – musimy zadeklarować wszystkie stałe jawne dyrektywą „=” zanim użyjemy tych stałych. Oczywiście zaraz po tym jak przedefiniujemy stałą symboliczną, poprzednia wartość tej stałej jest zapominana. Zauważmy, że nie możemy przedefiniować symboli zadeklarowanych dyrektywami textequ i equ.

8.6 DYREKTYWY PROCESORA

Domyślnie, MASM będzie assemblował tylko instrukcje które są dostępne na wszystkich członkach rodziny 80x86.W szczególności ,to znaczy, że nie będą asemblowane instrukcje które nie są dostępne na mikroprocesorach 8086 i 8088.Przez generowanie błędów dla instrukcji nie-8086,MASM zapobiega przypadkowemu zastosowaniu tych instrukcji nie są dostępne na różnych procesorach. Jest to dobre o ile, rzeczywiście chcemy użyć tych instrukcji dostępnych na procesorach poza 8086 i 8088.Dyrektywy procesora pozwalają nam zastosować instrukcje asemblacji dostępnych na późniejszych procesorów.

Dyrektywy procesora to:

.8086 .8087 .186 .286 .287 .286P .386 .387 .386P .486 .486P .586 .586P

Żadna z tych dyrektyw nie akceptuje żadnych operandów.

Dyrektywy procesora aktywują wszystkie instrukcje dostępne na danym procesorze. Ponieważ rodzina 80x86 jest kompatybilna w górę, wyszczególniając poszczególne dyrektywy procesora aktywując wszystkie instrukcje na tym procesorze i również wszystkich wcześniejszych procesorach.

Dyrektywy .8087, .287, i .387 aktywują zbiór instrukcji zmiennoprzecinkowych dla danego koprocesora zmiennoprzecinkowego. Jednakże dyrektywa .8086 również włącza zbiór instrukcji 8087;podobnie .286 aktywuje zbiór instrukcji 80286 a .386 aktywuje zbiór instrukcji zmiennoprzecinkowych 80387.jedynym celem dla tych dyrektyw FPU (koprocesor arytmetyczny) jest zezwolenie na współpracę instrukcji 80287 z 8087 lub zbioru instrukcji 80186 lub 80387 ze zbiorem instrukcji 8086,80186 lub 80286.

Dyrektywy procesora zakończone „P" pozwalają assemblować w trybie uprzywilejowanym instrukcji. Instrukcje trybu uprzywilejowanego są użyteczne tylko dla pisania systemów operacyjnych, pewnych sterowników i innych zaawansowanych podprogramów systemowych. Ponieważ ten tekst nie omawia instrukcji trybu uprzywilejowanego, omówimy trochę tych instrukcji później.

Procesory 80386 i późniejsze wspiera dwa typy segmentów kiedy działamy w trybie chronionym – 16 bitowe segmenty i 32 bitowe segmentowe. W trybie rzeczywistym. te procesory wspierają tylko segmenty 16 bitowe. Assembler musi wygenerować subtelnie różne opcody dla 16 i 32 bitowych segmentów .Jeśli wyszczególnimy 32 bitowy procesor używając .386,.486 lub .586,MASM domyślnie generuje instrukcje dla 32 bitowego segmentu. Jeśli spróbujemy uruchomić taki kod w trybie rzeczywistym pod MS-DOSem , prawdopodobnie spowodujemy krach systemu. Są dwa rozwiązania tego problemu. Pierwszym jest wyszczególnienie use16 jako operandu dla każdego segmentu jaki tworzymy w naszym programie .Drugie rozwiązanie jest odrobinę bardziej praktyczne, po prostu wstawimy następującą instrukcję po 32 bitowej dyrektywy procesora:

```
option          segment:use16
```

Ta dyrektywa mówi MASMowi aby wygenerował domyślnie segmenty 16 bitowe zamiast 32 bitowych segmentów.

Zauważmy, że MASM nie wymaga procesora 80486 lub Pentium jeśli wyszczególniamy dyrektywy .486 lub .586.Sam assembler jest napisany w kodzie 80386 (począwszy od wersji 6.1) więc potrzebujemy tylko procesora 80386 dla asemblacji każdego programu MASMem. Oczywiście, jeśli używamy określonych instrukcji procesorów 80486 lub Pentium, będziemy musieli zastosować procesor 80486 lub Pentium dla uruchomienia zassemblowanego kodu.

Możemy selektywnie aktywować lub dezaktywować różne zbiory instrukcji w całym naszym programie. Na przykład możemy włączyć instrukcje 80386 w kilku liniach naszego kodu a potem powrócić do instrukcji 8086.Następująca sekwencja to demonstruje:

```
.386                    ;zaczynamy używać instrukcji 80386
-
-                       ,ten kod może mieć instrukcje 80386
-
.8086                   ;wracamy do instrukcji 8086
-
-
-                       ;ten kod może mieć tylko instrukcje 8086
```

Możliwe jest napisanie programu, który wykrywa, podczas wykonywani programu, na jakim procesorze rzeczywiście jest wykonywany program. Dlatego też, możemy wykryć procesor 80386 i używać instrukcji 80386.Jeśli nie wykryjemy procesora 80386 ,możemy wetknąć instrukcje 8086.Poprzez selektywne włączanie instrukcji 80386 w tych segmentach naszego programu, który wykonujemy jeśli jest obecny procesor 80386,możemy wykorzystać dodatkowe instrukcje. Podobnie, poprzez wyłączanie zbioru instrukcji 80386 w inne sekcji naszego programu, możemy zabezpieczyć się przed nieumyślnym zastosowaniem instrukcji 80386 w części programu dla 8086.

8.7 PROCEDURY

W odróżnieniu od HLL'i, MASM nie wprowadza surowych zasad co do tego jak składać procedury. Możemy wywołać procedurę spod każdego adresu w pamięci. Pierwsza instrukcja ret napotkana podczas wykonywania programu kończy procedurę. Taka pełna wyrazu swoboda, często jest nadużywana przez programy, które są bardzo trudne do odczytu i pielęgnacji. Dlatego też MASM dostarcza udogodnień przy deklaracji procedur wewnątrz naszego kodu. Podstawowym mechanizmem dla deklaracji jest:

```
Procname        proc    {NEAR or FAR}
                <instrukcje>
procname        endp
```

Jak widzimy, definicja procedury wygląda podobnie do tej dla segmentu. Jedyna różnica jest taka ,że procname (to znaczy nazwa definiowanej procedury) musi być unikalnym identyfikatorem wewnątrz naszego programu. Nasz kod wywołuje tą procedurę używając jej nazwy, nie zrobi tego jeśli będzie miał inną procedurę o tej samej nazwie; jak program może określić który podprogram wywołać?

Proc może mieć kilka operandów, mimo, że możemy rozważać tylko trzy: pojedyncze słowo kluczowe near, pojedyncze słowo kluczowe far lub puste pole operandu. MASM używa tych operandów do określenia czy wywołaliśmy tą procedurę instrukcją bliskiego lub dalekiego wywołania. Określają one również jaki typ instrukcji ret MASM wyemituje wewnątrz procedury. Rozważmy następujące dwie procedury:

```
NProc       proc    near
            mov     ax, 0
            ret
NProc       endp

FProc       proc    far
            mov     ax, 0FFFFH
            ret
FProc       endp

i:
            call    NPROC
            call    FPROC
```

Assembler automatycznie generuje trzybajtowe (bliskie) wywołanie dla pierwszej instrukcji call, ponieważ wie ,że NProc jest procedurą bliską. generuje również pięciobajtową (daleką) instrukcję call dla drugiego wywołania, ponieważ FProc jest procedurą daleką. Wewnątrz samej procedury, MASM automatycznie konwertuje wszystkie instrukcje ret do bliskiego lub dalekiego powrotu w zależności od typu podprogramu.

Zauważmy, że jeśli nie zakończymy sekcji proc /endp ret'em lub inną instrukcją sterowania przesyłaniem danych a przebieg programu wykonuje się do dyrektywy endp, wykonywanie będzie kontynuowane od następnej wykonywalnej instrukcji następującej po endp .Na przykład rozważmy coś takiego:

```
Proc1       proc
            mov     ax, 0
Proc1       endp
Proc2       proc
            mov     bx, 0FFFFH
            ret
Proc2       endp
```

Jeśli wywołamy Proc1, sterowanie przejdzie do Proc2 z instrukcją mov bx, 0FFFFh.W odróżnieniu od procedur języków wysokiego poziomu, procedury język assemblera nie zwierają ukrytych instrukcji powrotu przed dyrektywą endp. Więc zawsze musimy być świadomi jak pracują dyrektywy proc / endp.

Nie jest nic specjalnego w deklaracjach procedur. Są one wygodą dostarczoną przez assembler, niczym więcej. Możemy pisać programy w języku assemblera prze resztę swojego życia i nigdy nie użyć dyrektyw proc i endp. Jednak robiąc tak, bylibyśmy kiepskimi programistami. Proc i endp są dobrze udokumentowanymi cechami, które, kiedy są użyte poprawnie, mogą pomóc nam uczynić nasze programy dużo łatwiejsze do odczytu i pielęgnacji.

MASM w wersji 6.0 i późniejszych traktuje wszystkie etykiety instrukcji wewnątrz procedury jako lokalne. To znaczy, nie możemy odwołać się bezpośrednio do tych symboli z zewnątrz procedury.

8.8 SEGMENTY

Wszystkie programy składają się z jednego lub więcej segmentów. Oczywiście, kiedy nasz program jest uruchomiony, rejestry segmentowe 80x86 wskazują na bieżący aktywny segment. W 80286 i wcześniejszych procesorach, możemy mieć do czterech aktywnych segmentów na raz. (kodu, danych, specjalny i stosu);w 80386 i późniejszych procesorach są dwa dodatkowe rejestry segmentowe s i gs. Chociaż nie możemy uzyskać dostępu do danych w więcej niż czterech lub sześciu segmentach w danej chwili, możemy modyfikować rejestry segmentowe 80x86 aby wskazywały na inne segmenty w pamięci pod kontrolą programu. To znaczy, że program może uzyskać dostęp do więcej niż czterech lub sześciu segmentów. Pytanie :"jak tworzymy te różne segmenty w programie i jak możemy uzyskać dostęp do nich w czasie wykonywania?".

Segmenty, w naszym asemblerowym pliku źródłowym, są definiowane dyrektywami segment i ends. Możemy wprowadzić tak dużo segmentów ile chcemy w naszym programie. Cóż, w rzeczywistości jesteśmy ograniczeni do 65,536 różnych segmentów przez procesory 80x86 a MASM prawdopodobnie nawet nie pozwoli

na tyle, ale prawdopodobnie nigdy nie będziemy przekraczać liczby segmentów na jakie pozwala nam MASM do wprowadzenia w naszym programie.

Kiedy MS-DOS zaczyna wykonywanie naszego ,inicjuje dwa rejestry segmentowe. Wskazuje cs jako segment zawierający nasz program główny i ss jako nasz segment stosu. Z tego punktu widzenia, jesteśmy odpowiedzialni za utrzymanie swoich rejestrów segmentowych.

Aby uzyskać dostęp do danych w szczególnym segmencie ,rejestry segmentowe 80x86 muszą zawierać adres tego segmentu. Jeśli uzyskujemy dostęp do danych w kilku różnych segmentach, nasz program będzie musiał załadować rejestr segmentowy adresem segmentowym przed uzyskaniem dostępu do niego. Jeśli często uzyskujemy dostęp do danych w różnych segmentach, spędzimy dużo czasu przeładowując rejestry segmentowe. Na szczęście, większość programów wykazuje lokalność odniesienia kiedy uzyskuje dostęp do danych. To znaczy, że kawałek kodu będzie uzyskiwał dostęp do tej samej grupy zmiennych wiele razy podczas danego okresu czasu .Łatwo jest zorganizować tak nasze programy, aby zmienne do których często chcemy uzyskać dostęp, pojawiły się w tym samym segmencie. Przez aranżowanie naszych w taki sposób, możemy zminimalizować liczbę razy, jakiej będziemy potrzebować. Do ładowania rejestrów segmentowych. W tym sensie, segment nie jest niczym więcej niż buforem dla często używanych danych.

W trybie rzeczywistym, segment może być długi na 65Kilobajty.Większość programów czystego języka assemblera używa mniej niż 64K kodu,64K danych globalnych i 64K przestrzeni stosu .Dlatego też możemy często korzystać z nie więcej niż trzech lub czterech segmentów w naszym programie. Faktycznie ,plik SHELL.ASM (zawierający szkielet programu asemblerowego) definiuje cztery segmenty a generalnie możemy użyć tylko trzech z nich. Jeśli użyjemy pliku SHELL.ASM jako podstawy naszych programów, rzadko będziemy musieli się martwić o dzielenie na segmenty w 80x86.Z drugiej strony, jeśli chcemy pisać złożone programy, będziemy musieli zrozumieć segmentację.

Segment w naszym pliku powinien przybrać formę:

```
segmentname        segment        {READONLY} {align} {combine} {use} {'class'}
                   Instrukcje
segmentname        ends
```

Następna sekcja opisuje każdy z tych operandów dyrektywy segment

Notka: segmentacja jest koncepcją, która dla wielu początkujących programistów asemblerowych wydaje się trudną do zrozumienia .Zauważmy ,że nie musimy zrozumieć dogłębnie segmentacji aby zacząć pisać programy asemblerowe 80x86.Jeśli będziemy robić kopię pliku SHELL.ASM do każdego programu jaki piszemy ,możemy zignorować sprawę segmentacji. Głównym zadaniem pliku SHELL.ASM jest zaopiekowanie się szczegółami segmentacji. Tak długo jak nie będziemy pisać niezmiernie długich programów lub używać olbrzymiej ilości danych, powinniśmy móc użyć SHELL.AM i zapomnieć o segmentacji. Pomimo to, ostatecznie możemy chcieć napisać większe programy asemblerowe, lub możemy chcieć napisać podprogram asemblerowy dla języka wysokiego poziomu takiego jak Pascal lub C++.W tym miejscu będziemy musieli poznać troszkę segmentację

8.8.1 NAZWY SEGMENTÓW

Dyrektywa segmentu wymaga etykiety w polu etykiety. Etykieta ta jest nazwą segmentu. MASM używa nazw segmentów dla trzech celów: łączenia segmentów ,określenia czy prefiks przesłonięcia segmentu jest konieczny uzyskanie adresu segmentu. Musimy również wyszczególnić nazwę segmentu w polu etykiety dyrektywy ends która kończy segment.

Jeśli nazwa segmentu nie jest unikalna (tj. zdefiniowaliśmy ją gdzieś w programie.),inne użycie musi również być definicją segmentu. Jeśli jest inny segment o tej samej nazwie, wtedy assembler traktuje tą definicję segmentu jako kontynuację poprzedniego segmentu używającego tej samej nazwy. Każdy segment ma swoją własną wartość licznika lokacji z nim powiązaną. Kiedy zaczynamy nawy segment(to znaczy, segment ,którego nazwa nie pojawiła się jeszcze w pliku źródłowym) MASM stworzy nową zmienną licznika lokacji, początkowo zero dla segmentu. Jeśli MASM napotka definicję segmentu która jest kontynuacją poprzedniego segmentu, wtedy MASM używa wartości licznika lokacji na końcu poprzedniego segmentu. Np.

```
CSEG               segment
                   mov     ax,bx
                   ret
CSEG               ends
DSEG               segment
Item1              byte    0
Item2              word    0
DSEG               ends
CSEG               segment
                   mov     ax,10
```

```
                add     ax,Item1
                ret
CSEG            ends
                End
```

Pierwszy segment (CSEG) zaczyna się wartością zero licznika lokacji .Instrukcja mov ax ,bx jest dwubajtowa a instrukcja ret jest jednobajtowa, więc licznik lokacji wynosi trzy na końcu segmentu. DSEG jest innym trzybajtowym segmentem, więc licznik lokacji powiązany z DSEG również zawiera trzy na końcu segmentu. Trzeci segment również ma tą samą nazwę jak segment pierwszy (CSEG),dlatego też assembler zakłada, że są one tym samym segmentem , w drugim wystąpieniu ,jako proste przedłużenie pierwszego .Dlatego też, kod umieszczony w drugim segmencie CSEG będzie assemblowany poczynając od offsetu trzy wewnątrz CSEG – faktyczne kontynuowanie kodu w pierwszym segmencie CSEG.

Kiedykolwiek wyszczególnimy nazwę segmentu jako operand instrukcji, MASM użyje bezpośredniego trybu adresowania i zastąpi adres tego segmentu dla jego nazwy. Ponieważ nie możemy załadować wartości bezpośredniej do rejestru segmentu pojedynczą instrukcją, ładujemy adres segmentowy do rejestru segmentowego dwoma instrukcjami. Na przykład, następujące trzy instrukcje pojawiają się na początku pliku SHELL.ASM, inicjują one rejestry ds i es więc wskazują one segment dseg:

```
        mov     ax, dseg        ;ładuje ax adres segmentowy dseg
        mov     ds.,ax          ;ds. wskazuje dseg
        mov     es,ax           ;es wskazuje dseg
```

Innym celem dla nazwy segmentu jest dostarczenie części składowej nazwie zmiennej. Pamiętamy ,adresy 80x86 zawierają dwie części składowe :segment i offset. Ponieważ 80x86 domyślnie większość danych odnosi do segmentu danych, jest powszechną praktyką wśród programistów asemblerowych, robić to samo, to znaczy nie zawracają sobie głowy wyszczególnianiem nazwy segmentu kiedy uzyskują dostęp do zmiennych w segmencie danych. Faktycznie, pełne odniesienie do zmiennej składa się z nazwy segmentu, dwukropka i nazwy offsetu:

```
        mov     ax, dseg : Item1
        mov     dseg : Item2, ax
```

Technicznie, powinniśmy wstawić przedrostek z nazwą segmentu przed wszystkimi zmiennymi w ten sposób. Jednak ,większość programistów nie martwi się tym specjalnym wymaganiem. Większość czasu będziemy się obchodzić bez tego ;jednak jest kilka momentów, kiedy rzeczywiście będziemy musieli wyszczególnić nazwę segmentu. Na szczęście, sytuacje te są rzadkie i tylko występują w bardzo złożonych programach.

Ważne jest ,żeby zdawać sobie sprawę, że wyspecyfikowanie nazwy segmentu przed nazwą zmiennej nie znaczy, że możemy uzyskać dostęp do danej w segmencie bez posiadania jakiegoś rejestru segmentowego wskazującego na ten segment. Z wyjątkiem instrukcji jmp i call nie ma instrukcji 80x86 które pozwalają nam wyspecyfikować pełny 32 bitowy bezpośredni adres segmentowy. Wszystkie inne odniesienia do pamięci używają rejestru segmentowego dla dostarczenia części składowych adresu segmentowego

---

8.8.2 PORZĄDEK ŁADOWANIA SEGMENTÓW

Normalnie segmenty są ładowane do pamięci w kolejności w jakiej pojawiają się w naszym pliku źródłowym W powyższym przykładzie, DOS będzie ładował segment CSEG do pamięci przed segmentem DSEG .Jeśli nawet segment CSEG pojawia się w dwóch częściach, przed i po DSEG, deklaracja CSEG przed wystąpieniem DSEG mówią DOSowi aby załadował cały segment CSEG do pamięci przed DSEG. Aby załadować DSEG przed CSEG, możemy użyć następującego programu:

```
DSEG    segment public
DSEG    ends
CSEG    segment public
        mov     ax ,bx
        ret
CSEG    ends
DSEG    segment public
Item1   byte    0
Item2   byte    0
DSEG    ends
CSEG    segment public
        mov     ax, 10
        add     ax, Item1
        ret
CSEG    ends
        end
```

Pusty segment deklarowany jako DSEG nie emituje żadnego kodu. Wartość licznika lokacji dla DSEG wynosi zero na końcu definicji segmentu. W związku z tym mam zero na początku następnego segmentu DSEG, dokładnie jakby był kontynuacją poprzedniej wersji programu. Jednak ponieważ deklaracja DSEG pojawia się pierwszy w programie, DOS załaduje go do pamięci jako pierwszy.

Porządek pojawiania się jest tylko jednym z czynników porządku ładowania. Na przykład, jeśli użyjemy dyrektywy .alpha, MASM będzie organizował segmenty alfabetycznie zamiast według pierwszeństwa pojawiania. Opcjonalne operandy dyrektyw segmentu również sterują porządkiem ładowania. Te operandy są tematem następnej sekcji.

8.8.3 OPERANDY SEGMENTU

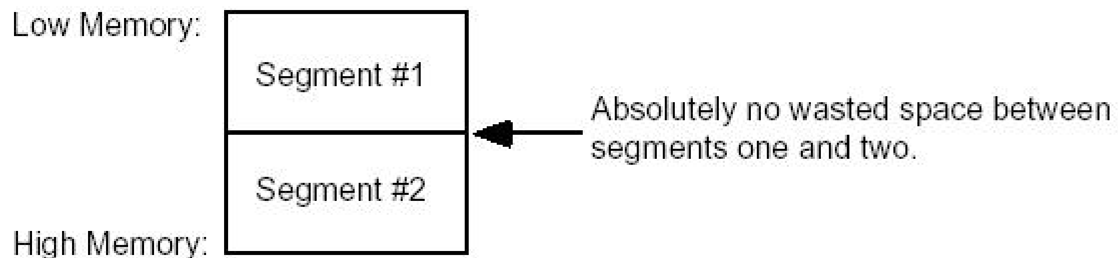
Dyrektywa segment pozwala na sześć różnych pozycji w polu operandu: operand wyrównania, operand łączenia, operand klasy, operand tylko do odczytu ‚operand „uses" i operand rozmiaru .Trzy z tych operandów sterują jak DOS ładuje segment do pamięci, pozostałe trzy sterują generowaniem kodu.

8.8.3.1 TYP WYRÓWNANIA

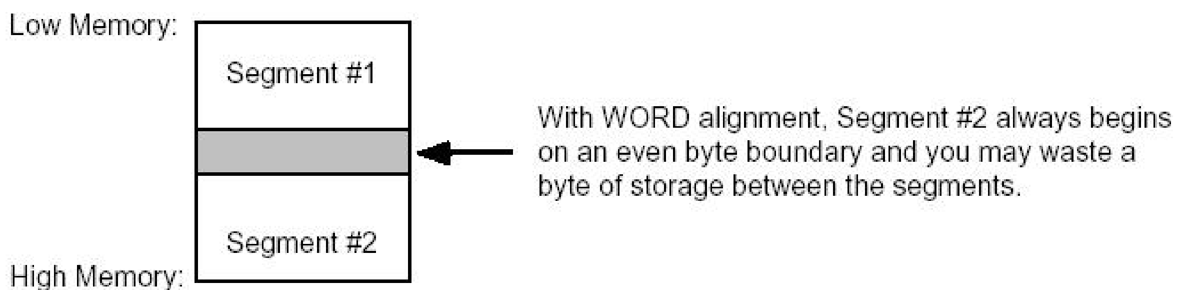
Parametr wyrównania jest jednym z następujących słów: byte, word, dword, para lub page. Te słowa kluczowe instruują assembler, linker i DOS do załadowania segmentu do granicy bajtu, słowa, podwójnego słowa, paragrafu lub strony. Parametr wyrównania jest opcjonalny .Jeśli jedno z powyższych słów kluczowych nie pojawia się jako parametr dyrektywy segmentu, domyślnym wyrównaniem jest paragraf (paragraf jest wielokrotnością 16 bajtów)

Wyrównywanie segmentu do granicy słowa ładuje segment do pamięci począwszy od pierwszego dostępnego bajtu po ostatnim segmencie. Wyrównanie do granicy słowa zaczyna segment od pierwszego bajtu parzystego adresu po ostatnim segmencie. Wyrównanie od granicy dword lokuje bieżący segment od pierwszego adresu który jest parzystą wielokrotnością cztery po ostatnim segmencie.

Na przykład, jeśli segment #1 jest zadeklarowany pierwszy w naszym pliku źródłowym a segment #2 bezpośrednio po nim i jest wyrównany bajtem, segment będzie przechowywany w pamięci jak następuje (zobacz rysunek 8.1)



Rysunek 8.1:Segment wyrównany bajtem



Rysunek 8.2: Segment wyrównany słowem

```
seg1        segment
            -
            -
            -
seg1        ends
seg2        segment     byte
            -
```

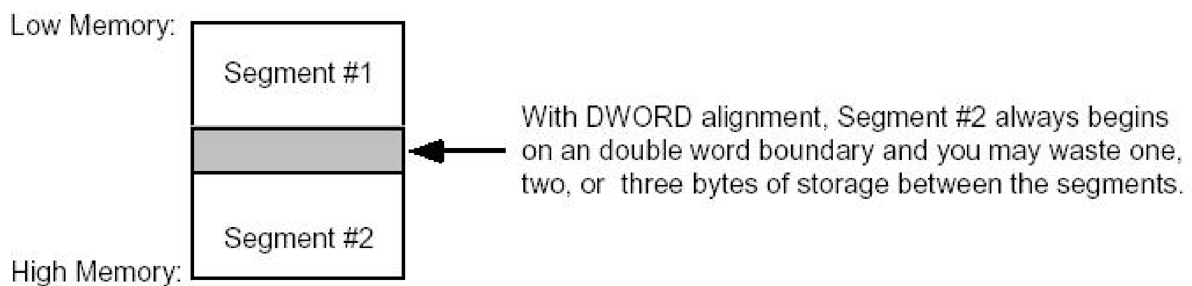
```
                        -
                        -
seg2                    ends
```

Jeśli segmenty jeden i dwa są zadeklarowane tak jak poniżej ,a segment #2 jest wyrównany słowem ,segmenty pojawią się w pamięci jak pokazano na rysunku 8.2

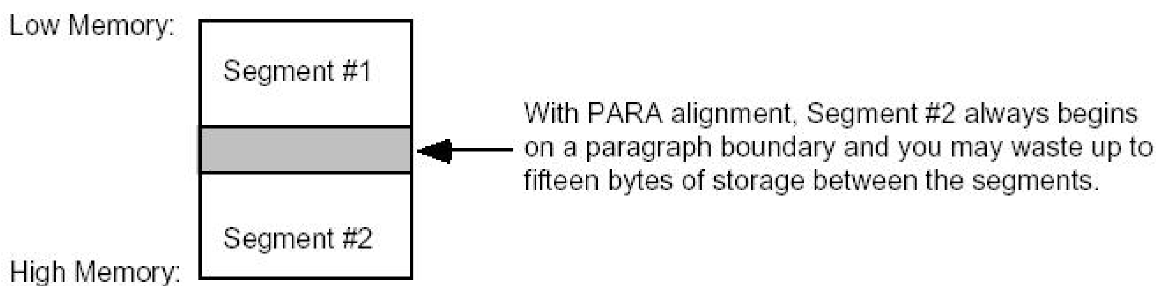
```
seg1        segment
            -
            -
            -
seg1        ends
seg2        segment     word
            -
            -
            -
seg2        ends
```

Inny przykład: jeśli segmenty jeden i dwa są takie jak poniżej. A segment #2 jest wyrównany podwójnym słowem, segmenty będą przechowywane w pamięci tak jak pokazano na rysunku 8.3

```
seg1        segment
            -
            -
            -
seg1        ends
seg2        segment     dword
            -
            -
            -
seg2        ends
```



Rysunek 8.3: Segment wyrównany podwójnym słowem



Rysunek 8.4: Segment wyrównany Paragrafem

Ponieważ rejestry segmentowe 80x86 zawsze wskazują adres paragrafu, większość segmentów jest wyrównywanych do 16 bajtowej granicy paragrafu (para).Przeważnie nasze segmentu powinny zawsze być wyrównywane do granicy paragrafu, chyba, że mamy dobry powód innego wyboru.

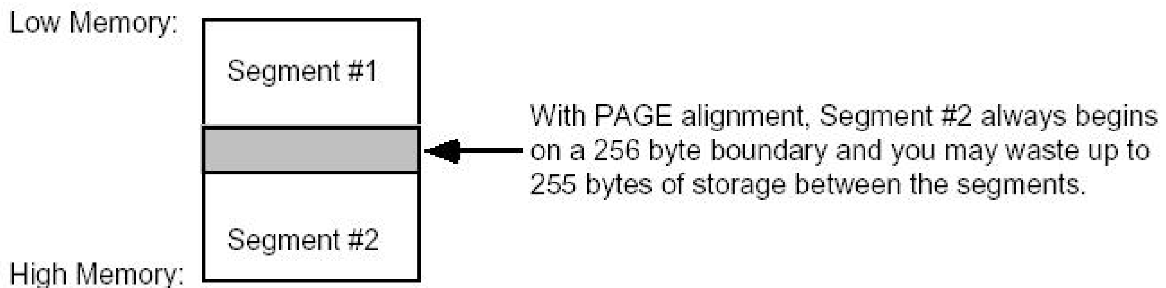
Na przykład, jeśli segmenty jeden i dwa są zadeklarowane tak jak poniżej ,a segment #2 jest wyrównany paragrafem, DOS będzie przechowywał segment w pamięci jak pokazano na rysunku 8.4

```
seg1        segment
            -
            -
            -
seg1        ends
seg2        segment      para
            -
            -
            -
seg2        ends
```

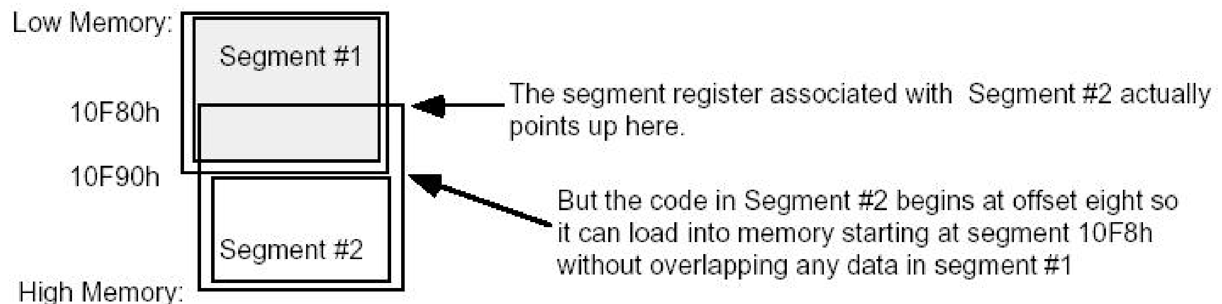
Granica strony wymusza wyrównanie segmentu zaczynającego się od następnego adresu który jest równy wielokrotności 256 bajtów. Pewne bufory danych mogą wymagać wyrównania do granicy 256 (lub 512) bajtów. Opcja wyrównania strony może być w tej sytuacji użyteczna.

Na przykład, jeśli segmenty jeden i dwa są zadeklarowane jak poniżej a segment #2 jest wyrównany stroną ,segmenty będą przechowywane w pamięci jak pokazano na rysunku 8.5

```
eg1         segment
            -
            -
            -
seg1        ends
seg2        segment      page
            -
            -
            -
seg2        ends
```



Rysunek 8.5: Segment z wyrównaniem strony



Rysunek 8.6: Wyrównanie segmentu do paragrafu

Jeśli wybierzemy jakieś wyrównanie inne niż bajt, assembler, linker i DOS mogą wprowadzić kilka fikcyjnych bajtów pomiędzy dwa segmenty ,żeby segment był właściwie wyrównany. Ponieważ rejestry segmentowe muszą zawsze wskazywać adres paragrafu(to znaczy, muszą być wyrównane paragrafem) możemy się zastanawiać jak procesor może adresować segment, który jest wyrównany do granicy bajtu, słowa lub podwójnego słowa. To jest łatwe. Kiedykolwiek wyszczególnimy wyrównanie segmentu, które wymusza segment zaczynający się spod adresu, który nie jest granicą paragrafu, assembler /linker założą, że rejestr

segmentowy wskazuje na poprzedni adres paragrafu a licznik lokacji zaczyna się jakimś offsetem do tego segmentu innego niż zero. Na przykład przypuśćmy ,że segment #1 kończy się na adresie fizycznym 10F87h a segment #2 jest wyrównany bajtem. Kod dla segmentu #2 będzie się zaczynał pod adresem segmentowym 10F80h.Jednakże,będzie nachodził na segment #1 na osiem bajtów. Aby przezwyciężyć ten problem, licznik lokacji dla segmentu #2 będzie zaczynał się od 8,więc segment będzie ładowany do pamięci tuż poza segmentem #1.

Jeśli segment #2 jest wyrównany bajtem a segment #1 nie kończy się na parzystym adresie paragrafu ,MASM poprawi pozycję startową licznika lokacji dla segmentu #2,aby mógł użyć poprzedniego adresu paragrafu przy dostępie do niego. (zobacz Rysunek 8.6)

Ponieważ 80x86 wymaga wszystkich segmentów do rozpoczęcia granicy paragrafu w pamięci, MASM (domyślnie) zakłada, że chcemy wyrównania paragrafem dla naszego segmentu. Poniższa definicja segmentu jest zawsze wyrównywana do granicy paragrafu:

```
CSEF        segment
            mov         ax,bx
            ret
CSEG        ends
            end
```

8.8.3.2 TYP ŁĄCZENIA

Typ łączenia steruje porządkiem w jakim segmenty o tej samej nazwie są wypisywane w kodzie wynikowym pliku tworzonego przez assembler. Dla wyspecyfikowania typu łączenia używamy jedno ze słów kluczowych public, stack, common, memory lub at .Memory jest synonimem dla public utrzymywane z powodu kompatybilności; powinniśmy zawsze używać public zamiast memory. Common i at są zaawansowanymi typami łączenia, które nie będą rozpatrywane w tym tekście. Typ łączenia stack powinien być używany z naszym segmentem stosu. Typ łączenia public powinien być używany wszędzie indziej.

Typy łączenia public i stack w gruncie rzeczy wykonują te same operacje. Łączą one segmenty o tej samej nazwie do pojedynczego ,ciągłego segmentu, jak opisano wcześniej. Różnice pomiędzy nimi jest sposób w jaki DOS manipuluje inicjacją rejestrów segmentu stosu i wskaźnikiem stosu. Wszystkie programy powinny mieć przynajmniej jeden typ stack (albo linker wygeneruje ostrzeżenie);reszta powinna być publiczna. MS-DOS automatycznie wskazuje rejestr segmentu stosu w segmencie zadeklarowanym typem łączenia stack kiedy ładuje program do pamięci.

Jeśli nie wyszczególnimy typu łączenia. wtedy assembler nie połączy segmentów kiedy tworzy plik kodu wynikowego praktyce, brak jakiegoś słowa kluczowego typu łączenia tworzy domyślnie typ łączenia private. Chyba, że typy klas są takie same (zobacz następną sekcję),każdy segment będzie emitowany jeśli MASM napotka go w pliku źródłowym. Na przykład rozważmy następujący program:

```
CSEG        segment     public
            mov         ax, 0
            mov         VAR1, ax
CSEG        ends
DSEG        segment             public
I           word                ?
DSEG        ends
CSEG        segment             public
            mov                 bx, ax
            ret
CSEG        ends
DSEG        segment             public
J           word                ?
DSEG        ends
            end
```

Ta część programu przynosi ten sam kod co:

```
CSEG                segment     public
                    mov         ax, 0
                    mov         VAR1, ax
                    mov         bx,ax
                    ret
CSEG                ends
DSEG                segment     public
I                   word        ?
J                   word        ?
```

```
DSEG            ends
                end
```

Assembler automatycznie łączy wszystkie segmenty które mają takie same nazwy i są publiczne .Przyczyną dla której assembler pozwala nam oddzielać segmenty jak te ,jest wygoda. Przypuśćmy, że mamy kilka procedur, z których każda wymaga pewnych zmiennych. Możemy zadeklarować wszystkie zmienne gdzieś w jednym segmencie, ale jest to często zakłóceniem. Większość ludzi jednak deklaruje swoje zmienne przed procedurą która ich używa. Poprzez używanie typu łączeniowego public z deklarowaniem segmentu, możemy zadeklarować nasze zmienne przed użyciem ich a assembler automatycznie przesunie te zmienne do właściwego segmentu kiedy assembluje program. Na przykład

```
CSEG            segment     public
;To jest procedura #1
DSEG            segment     public
;lokalne zmienne dla procedury #1
VAR1            word        ?
DSEG            ends
                mov         AX,0
                mov         VAR1,AX
                mov         BX,AX
                ret
;To jest procedura #2
DSEG            segment     public
I               word        ?
J               word        ?
DSEG            ends
                mov         ax,I
                add         ax,J
                ret
CSEG            ends
                ens
```

Zauważmy ,że możemy zagnieździć segmenty w zadawalający sposób Niestety w MASM zakres działania zmiennych nie pracuje w ten sam sposób jak w HLL'ach takich jak Pascal .Normalnie ,raz zdefiniowany symbol wewnątrz naszego programu, jest widzialny gdziekolwiek w programie.

8.8.4 TYP KLASY

Końcowym operandem dyrektywy segment jest zazwyczaj typ klasy. Typ klasy wyszczególnia porządek segmentów, które nie mają takich sam nazw segmentów. Ten operand składa się z symbolu zamkniętego przez apostrofy (cudzysłów nie jest tam dozwolony).generalnie powinniśmy używać następujących nazw: DODE (dla segmentu zawierającego kod programu);DATA(dla segmentu zawierającego zmienne, dane stałe i tablice);CONST (dla segmentu zawierającego dane stałe i tablice);i STACK (dla segmentu stosu),Poniższy fragment programu ilustruje ich użycie:

```
CSEG            segment     public 'CODE'
                mov         ax, bx
                ret
CSEG            ends

DSEG            segment     public 'DATA'
Item1           byte        0
Item2           byte        0
DSEG            ends
CSEG            segment     public 'CODE'
                mov         ax, 10
                add         Ax, Item1
                ret
CSEG            ends

SSEG            segment     stack 'STACK'
STK             word        4000  dup (?)
SSEG            ends
```

```
C2SEG        segment      public „CODE'
             ret
C2SEG        ends
             end
```

Rzeczywiste ładowanie procedury jest realizowane jak następuje. Assembler lokuje pierwszy segment w pliku{Ponieważ jest on segmentem łączonym public, MASM łączy wszystkie inne segmenty CSEG na końcu tego segmentu. W końcu ponieważ jest łączona klasa 'CODE',MASM dołączy wszystkie segmenty (C2SEG) o tych samych nazwach klas później. Po działaniu na tych segmentach, MASM szuka pliku źródłowego Dla następnego niepołączonego segmentu i powtarza cały proces. W powyższym przykładzie ,segmenty będą ładowane w następującym porządku: CSEG, CSEG(drugie wystąpienie),C2SEG.DSEG a potem SSEG. Ogólna zasada dotycząca jak nasz plik będzie ładowany do pamięci jest następująca

- (1)Assembler łączy wszystkie segmenty publiczne, które mają taką samą nazwę
- (2)Już połączone, segmenty są wysyłane do pliku z kodem wynikowym w porządku swojego pojawiania się w pliku źródłowym. Jeśli nazwa segmentu pojawia się dwa razy wewnątrz pliku źródłowego (a jest public),wtedy segment łączony będzie wysyłany do pliku z kodem wynikowym na pozycji określonej przez pierwsze wystąpienie segmentu wewnątrz pliku źródłowego.
- (3)Linker odczytuje plik z kodem wynikowym stworzony przez assembler i przestawia segmenty kiedy tworzy plik wykonywalny. Linker zaczyna przez przypisanie pierwszego znalezionego segmentu w pliku kodu wynikowego do pliku .EXE. Potem szuka w całym pliku z kodem wynikowym każdego segmentu z taką samą nazwą klasy. Takie segmenty są sekwencyjnie przypisywane do pliku .EXE.
- (4)Kiedy wszystkie segmenty o tej samej nazwie klas jak segment pierwszy zostały wyemitowane do pliku .EXE, linker szuka pliku z kodem wynikowym dla następnego segmentu który, nie należy do tej samej klasy co poprzedni segment (y).Zapisuje ten segment do pliku .EXE a i powtarza krok (3) dla każdego segmentu należącego do tej klasy.
- (5)W końcu, linker powtarza krok (4) aż do chwili kiedy zlinkuje wszystkie segmenty w pliku z kodem wykonawczym.

8.8.5 OPERAN TYLKO DO ODCZYTU

Jeśli tylko do odczytu jest pierwszym operandem dyrektywy segment, assembler będzie generował błąd jeśli napotka jakąś instrukcję, która próbuje zapisać coś do tego segmentu. Jest to bardzo użyteczne dla kodu segmentu. Opcja ta w rzeczywistości nie zapobiega zapisaniu do tego segmentu w czasie wykonania. Jest bardzo łatwa sztuczka assemblera aby zapisać do tego segmentu tak czy tak.. Jednakże, przez wyszczególnienie readonly, możemy wyłapać jakieś powszechne błędy programistyczne ,które inaczej moglibyśmy przegapić. Ponieważ, rzadko będziemy umieszczać zmienne programowalne w naszym segmencie kodu, jest prawdopodobnie dobrym pomysłem ,uczynić nasze segmenty kodu readonly.

Przykład operandu READONLY:

```
seg1      segment      readonly para public  'DATA'
          -
          -
          -
seg1      ends
```

8.8.6 OPCJE USE16,USE32 I FLAT

Kiedy pracujemy z procesorem 80386 lub późniejszym, MASM generuje różne kody dla 16 a 32 bitowych segmentów. Kiedy piszemy kod do wykonywania w trybie rzeczywistym pod DOS, musimy zawsze używać segmentów 16 bitowych.32 bitowe segmenty mają zastosowanie do programów uruchomionych w trybie chronionym. Niestety MASM często domyślnie ustawia tryb 32 bitowy, kiedy tylko wybierzemy procesor 80386 lub późniejszy używając dyrektywy .386,.486 lub .586 w programie. Jeśli chcemy użyć 32 bitowych instrukcji, będziemy musieli jasno powiedzieć MASMowi o użyciu 16 bitowych segmentów. Operandy use16,use32 i flat dyrektywy segment pozwalają nam wyszczególnić rozmiar segmentu. Dla większości programów DOS'owskich ,zawsze będziemy używać operandu use16.Mówi on MASMowi ,że segment jest segmentem 16 bitowym i aby zasemblował go odpowiednio. Jeśli używamy jednej z dyrektyw do aktywacji zbioru instrukcji 80386 lub późniejszych, powinniśmy użyć use16 w całym naszym segmencie kodu lub MASM wygeneruje zły kod.

Przykład użycia operandu use16:

```
seg1      segment      para public use16  'data'
          -
          -
          -
```

```
seg1            ends
```

Operandy use32 i flat mówią MASMowi aby wygenerował kod dla 32 bitowego segmentu. Ponieważ ten tekst nie zajmuje się programowanie w trybie chronionym, nie będziemy rozważać tych opcji.

Jeśli chcemy wymusić use16 jako domyślny w programie, który zezwala na instrukcje 80386 i późniejsze, jest jeden sposób na zrobienie tego. Umieszczenie następującej dyrektywy w naszym programie przed każdym segmentem:

```
.option         segment:use16
```

8.8.7 DEFINICJE TYPOWYCH SEGMENTÓW

Czy powyższe omówienie pozostawiło cię całkiem zakłopotanym? Nie martw się tym. Dopóki nie będziemy pisali wyjątkowo długich programów, nie będziemy musieli martwić się operandami powiązanymi z dyrektywą segment. Dla większości programów, następujące trzy segmenty powinny okazać się wystarczające:

```
DSEG            segment         para public 'DATA'
;tu wprowadzamy definicje naszych zmiennych
DSEG            ends

CSEG            segment         para public use16 'CODE'
;tu wprowadzamy instrukcje naszego programu
CSEG            ends
SSEG            segment         para stack 'STACK'
stk             word            1000h dup (?)
EndStk          equ             this word
SSEG            ends
                end
```

Plik SHELL.ASM automatycznie deklaruje te trzy segmenty. Jeśli zawsze będziemy kopiować pliki SHELL.ASM przy pisaniu nowego programu prawdopodobnie nie będziemy musieli martwić się o deklaracje segmentów i segmentację.

8.8.8 DLACZEGO BĘDZIEMY CHCIELI STEROWAĆ PORZĄDKIEM ŁADOWANIA

Pewne wywołania DOS wymagają aby podać długość naszego programu jako parametr. Niestety, obliczanie długości programu zawierającego kilka segmentów jest bardzo trudnym procesem. Jednakże ,kiedy DOS ładuje nasz program do pamięci, załaduje cały program do zawartego bloku RAMu. Dlatego do obliczanie długości programu potrzebujemy znać tylko adres startowy i końcowy naszego programu .Po prostu różnica tych dwóch wartości stanowi o długości naszego programu.

W programie, który zawiera wiele segmentów, będziemy musieli znać który segment będzie ładowany jako pierwszy, a który ostatni, żeby obliczyć długość naszego programu. Okazuje się, że DOS zawsze ładuje Prefiks Segmentu Programu lub PSP do pamięci, przed pierwszym segmentem naszego programu. Musimy brać pod uwagę długość PSP kiedy obliczamy długość programu. MS-DOS odkłada adres segmentowy PSP w rejestrze ds. Więc obliczenie różnicy pomiędzy ostatnim bajtem w programie a PSP da nam długość naszego programu. Następujący kod oblicza długość programu w paragrafie:

```
CSEG                    segment         public 'CODE'
                        mov             ax, ds.              ;pobierz adres segmentowy PSP
                        sub             ax, seg LASTSEG      ;oblicza różnicę
;AX zawiera teraz długość tego programu (w paragrafie)
                        -
                        -
                        -
CSEG                    ends
;Wstawiamy wszystkie inne segmenty tutaj.
LASTSEG                 segment         para public „LASTSEG'
LASTSEG                 ends
                        end
```

8.8.9 PRZEDROSTKI SEGMENTU

Kiedy 80x86 odnosi się do operandu pamięci, zazwyczaj odnosi się do lokacji wewnątrz bieżącego segmentu danych .jednak, możemy poinstruować mikroprocesor do odniesienia danych w jednym z innych segmentów używając przedrostka segmentu przed wyrażeniem adresowym.

Prefiksem segmentu jest ds:,cs:,ss:,es:,fs:, lub gs: .Kiedy używamy go przed wyrażeniem adresowym, prefiks segmentu instruuje 80x86 do pobrania operandu pamięci z wyszczególnionego segmentu zamiast segmentu domyślnego. na przykład, mov ax ,cs:I[bx] ładuje akumulator adresem I+bx wewnątrz bieżącego

segmentu kodu. Jeśli prefiks cs: byłby nieobecny, instrukcja normalnie załadowała by dane z bieżącego segmentu danych. Podobnie mov ds:[bp], ax przechowa akumulator w komórce pamięci wskazywanej przez rejestr bp w bieżącym segmencie danych (pamiętajmy ,kiedykolwiek używamy bp jako rejestry bazowego, wskazuje on na segment stosu).

Prefiksy segmentu są opcodami instrukcji. Dlatego też, kiedykolwiek ich używamy, zwiększamy długość (i zmniejszamy szybkość) instrukcji wykorzystujących prefiksu segmentu. Dlatego też, nie chcemy używać prefiksów segmentów chyba, że mamy ku temu dobry powód.

8.8.10 SEGMENTY STERUJĄCE DYREYKTYWY ASSUME

80x86 generalnie odnoszą się do pozycji danych w stosunku do rejestru segmentowego ds (lub segmentu stosu).Podobnie, wszystkie odwołania kodu (skoki, wywołania ,itp.) są zawsze w stosunku do bieżącego segmentu kodu. Jest tylko jedna kwestia – jak assembler wie, który segment to segment danych a który jest segmentem kodu (lub innym segmentem)?Dyrektywa segment nie mówi nam jakiego typu segment pojawi się w programie. Pamiętajmy ,segment danych jest segmentem danych ponieważ wskazuje na niego rejestr ds. Ponieważ rejestr ds może być zmieniony w czasie wykonywania programu (przy użyciu instrukcji jak mov ds., ax),każdy segment może być segmentem danych. Ma to pewne interesujące konsekwencje dla assemblera. Kiedy wyszczególniamy segment w naszym programie, nie tylko musimy powiedzieć CPU, że segment jest segmentem danych, ale musimy również powiedzieć asemblerowi gdzie i kiedy ten segment jest segmentem danych (lub kodu. stosu /specjalnym/ F/G).Dyrektywa ASSUME dostarcza tej informacji assemblerowi.

Dyrektywa assume przybiera następującą formę:

```
assume   {CS:seg} {DS.:seg} {ES:seg} {FS:seg} {GS:seg} {SS:seg}
```

Nawiasy klamrowe otaczające pozycje opcjonalne, nie są częścią tych operandów. Zauważmy ,że musi być przynajmniej jeden operand. Seg jest albo nazwą segmentu (zdefiniowaną dyrektywą segment) albo zarezerwowanym słowem nothing. Wiele operandów w polu operandu dyrektywy assume musi być oddzielonych przecinkiem. Przykłady poprawnych dyrektyw assume:

```
assume      DS.:DSEG
assume      CS:CSEG, DS.:DSEG, ES:DSEG, SS:SSEG
assume      CS:CSEG,DS.:NOTHING
```

Dyrektywa assume wskazuje assemblerowi, że musimy załadować wyszczególniony rejestr(y) segmentowe wyspecyfikowaną wartością adresu segmentowego. Zauważmy ,że ta dyrektywa nie modyfikuje żadnego rejestru segmentowego, po prostu mówi assemblerowi, żeby założył rejestry segmentowe wskazujące pewne segmenty w programie. Podobnie jak dyrektywy wyboru procesor i równości, dyrektywa assume modyfikuje zachowanie assemblera od gdzie napotyka ją MASM, aż do innej dyrektywy assume zmieniając podane założenia.

Rozważmy następujący program:

```
DSEG1       segment     para  public  'DATA'
var1        word        ?
DSEG1       ends

DSEG2       segment     para public  'DATA'
var2        word        ?
DSEG2       ends

CSEG        segment     para  public  'CODE'
            assume      CS:CSEG, DS.:DSEG1, ES:DSEG2
            mov         ax, seg DSEG1
            mov         ds.,ax
            mov         ax, seg DSEG2
            mov         es, ax
            mov         var1, 0
            mov         var2, 0
            -
            -
            -
            assume      DS.:DSEG2
            mov         ax, seg DSEG2
            mov         ds., ax
            mov         var2, 0
            -
```

```
                -
                -
CSEG            ends
                end
```

Kiedykolwiek assembler napotka nazwę symboliczną, sprawdza ,który segment zawiera ten symbol .W powyższym programie,var1 pojawia się w segmencie DSEG1 a var2 pojawia się w segmencie DSEG2.Pamiętamy,że mikroprocesor 80x86 nie wie o zadeklarowanych segmentach wewnątrz programu, może uzyskać dostęp do danych wskazywanych przez rejestry segmentowe cs,ds.,es,ss,fs i gs .Instrukcja assume w tym programie mówi assemblerowi, że rejestr ds. wskazuje na DSEG1 w pierwszej części programu, a DSEG2 w drugiej części programu.

Kiedy assembler napotyka instrukcję w postaci mov var1,0,pierwszą rzeczą jest określenie segmentu var1.Potem porównuje ten segment ponownie z listą uczynionych przez assembler założeń co do rejestrów segmentowych. Jeśli nie zadeklarujemy var1 w jednym z tych segmentów, wtedy assembler wygeneruje błąd stwierdzający, że program nie może uzyskać dostępu do tej zmiennej .Jeśli symbol (var1 w naszym przykładzie) pojawia się w jednym obecnie założonym segmencie, wtedy assembler sprawdza czy jest to segment danych. Jeśli tak, wtedy instrukcja jest assemblowana jak opisano w dodatkach. Jeśli symbol pojawia się w segmencie innym niż ten ,który zakłada assembler ,że wskazuje ds. ,wtedy assembler emituje bajt przedrostka przesłonięcia segmentu, specyfikujący faktyczny segment który zawiera dane.

W przykładowym powyższym programie, MASM będzie assemblował pierwsze wystąpienie instrukcji mov var2,0 z bajtem prefiksu segmentu es: ponieważ assembler, zakłada że es zamiast ds wskazuje DSEG2.MASM będzie assemblował drugie wystąpienie tej instrukcji bez bajtu prefiksu segmentu es ponieważ assembler zakłada ,że ds. wskazuje na DSEG2.Zapamiętajmy,ze jest bardzo i łatwo zmylić assembler. Rozpatrzmy następujący kod:

```
CSEG            segment         para public ‘CODE’
                assume          CS:CSEG,DS:DSEG1,ES:DSEG2
                mov             ax, seg DSEG1
                mov             ds,ax
                -
                -
                -
                jmp             SkipFixDS
                assume          DS.:DSEG2
FixDS:          mov             ax, seg DSEG2
                mov             ds, ax
SkipFixDS:
                -
                -
CSEG            ends
                End
```

Zauważmy, że ten program skacze do kodu który ładuje rejestr ds wartością segmentu DSEG2.To znaczy, że pod etykietą SkipFixDS rejestr ds zawiera wskaźnik do DSEG1 ,nie dSEG2.Jednakże assembler nie jest dość błyskotliwy dla rozwiązania tego problemu, więc ślepo zakłada, że ds wskazuje na DSEG2 zamiast na DSEG1To będzie klęska .Ponieważ assembler zakłada. że uzyskujemy dostęp do zmiennych w DSEG2 podczas gdy rejestr ds w rzeczywistości wskazuje na DSEG1,taki dostęp będzie się odnosił do komórki pamięci w DSEG1 pod taki sam offset jaki ma zmienna w DSEG2.Zmieni to dane w DSEG1 (lub nasz program będzie czytał nieprawidłowe wartości dla zmiennych założonych w segmencie DSEG2).

Dla początkujących programistów najlepszym rozwiązaniem tego problemu jest unikanie stosowania wielu segmentów (danych) wewnątrz swoich programów jeśli tylko to możliwe. Zachowajmy dostęp do wielu segmentów do dnia kiedy będziemy gotowi do zajęcia się takim problemem jak ten. Jako początkujący programista asemblerowy po prostu używaj jednego segmentu kodu, jednego segmentu danych i jednego segmentu stosu i pozostaw rejestry segmentowe wskazujące każdy na ten segment do którego jest przeznaczony. Dyrektywa assume jest całkiem złożona i może być przyczyną niezłych kłopotów jeśli będzie niewłaściwie używana. Lepiej nie zawracać sobie głowy używaniem assume do chwili kiedy lepiej zapoznamy się z całą ideo programowania w języku assemblera i segmentacji w 80x86.

Słowo zarezerwowane nothing mówi assemblerowi, że nie mieliśmy najmniejszego pojęcia gdzie ma wskazywać rejestr segmentowy. Mówi również assemblerowi ,że nie powinien uzyskiwać dostępu do danych w stosunku do tego rejestru segmentowego ,chyba, że wyraźnie dostarczy prefiks segmentu do adresu. Powszechną konwencją programistyczna jest umiejscawianie dyrektywy assume przed wszystkimi procedurami w programie.

Ponieważ wskaźnik segmentu deklarujący segment w programie rzadko zmienia punkty wejścia i wyjścia procedury, jest to idealne miejsce do wstawienia dyrektywy assume:

```
                assume      ds:P1Dseg, cs:cseg, es:nothing
Procedura1      proc        near
                push        ds              ;zachowanie ds
                push        ax              ;zachowanie ax
                mov         ax, P1Dseg      ;pobiera wskaźnik do P1dseg
                mov         ds, ax          ;do rejestry ds
                -
                -
                -
                pop         ax              ;przywrócenie wartości ax
                pop         ds              ;przywrócenie wartość ds
                ret
Procedura1      endp
```

Jedyne problem z tym kodem jest to ,że MASM jeszcze zakłada, że ds wskazuje P1Dseg kiedy napotka kod po Procedura1Najlepszym rozwiązaniem jest wprowadzenie drugiej dyrektywy assume po dyrektywie endp ,która powie MASMowi, że nic nie wie o wartości w rejestrze ds:

```
                -
                -
                -
                ret
Procedura1      endp
                assume      ds:nothing
```

Chociaż następna instrukcja w programie prawdopodobnie będzie inną dyrektywą assume daną przez assembler jako nowe założenie co do ds (na początku procedury, która następuje powyżej),jest lepszym pomysłem zaadoptować tą koncepcję. Jeśli opuścimy wprowadzenie dyrektywy assume przed następną procedurą w naszym pliku źródłowym, instrukcja assume ds:nothing będzie utrzymywała assembler w założeniu, że możemy uzyskać dostęp do zmiennych w P1Dseg.

Prefiks przesłonięcia segmentu zawsze ignoruje założenia czynione przez assembler. Mov ax, cs:var1 zawsze ładuje rejestr ax słowem spod offsetu var1 wewnątrz bieżącego segmentu kodu ,bez względu na to gdzie jest zdefiniowane var1.Głównym celem prefiksów przesłonięcia segmentów jest manipulacja pośrednimi odniesieniami. Jeśli mamy instrukcję w postaci mov ax,[bx] assembler zakłada, że bx wskazuje na segment danych. Jeśli rzeczywiście musimy uzyskać dostęp w różnych segmentach, możemy użyć przesłonięcia segmentu, w ten sposób mov ax, es:[bx].

Generalnie, jeśli mamy zamiar użyć wielu segmentów danych wewnątrz naszego programu powinniśmy użyć pełnej nazwy segment:offset dla naszej zmiennej. Np. mov ax,DSEG1:I i mov bx,DSEG2:J.Nie eliminuje to potrzeby ładowania rejestrów segmentowych lub uczynienia właściwego użytku z dyrektywy assume, ale uczyni to nasz program łatwiejszy do odczytu i pomoże MASMowi zlokalizować możliwe błędy w naszym programie.

Dyrektywa assume jest w rzeczywistości całkiem użyteczna dla innych spraw poza ustawieniem segmentu domyślnego. Zobaczymy więcej zastosowań dla tej dyrektywy trochę później w tym rozdziale.

8.8.11 SEGMENTY ŁĄCZONE: DYREKTYWA GROUP

Większość segmentów w typowym programie asemblerowym jest mniejsza niż 64 Kilobajty. Istotnie, większość segmentów jest dużo mniejszych niż 64KB.Kiedy MS-DOS ładuje segment programu do pamięci, kilka z segmentów może znaleźć się w pojedynczym 64KB regionie pamięci. W praktyce, możemy łączyć te segmenty w pojedynczym segmencie w pamięci. Może to umożliwić poprawę wydajności naszego kodu.

Więc dlaczego po prostu nie połączyć takich segmentów w naszym assemblerowym kodzie? Cóż, jak wskazała poprzednia sekcja, utrzymanie oddzielnych segmentów może pomóc nam skonstruować nasze programy lepsze i pomóc uczynić je bardziej modularne. Ta modularność jest bardzo ważna w naszych programach. Jak zwykle, poprawa struktury i modularności naszych programów może spowodować, że staną się mniej wydajne. Na szczęście, MASM dostarcza dyrektywy group, która pozwala nam traktować dwa segmenty jako ten sam segment fizyczny bez porzucania struktury i modularności naszego programu.

Dyrektywa group pozwala nam tworzyć nową nazwę segmentu, który obejmuje segmenty zgrupowane razem. Na przykład, jeśli mamy dwa segmenty nazwane „Module1Data" i „Module2Data",które życzymy sobie połączyć do pojedynczego segmentu fizycznego, możemy użyć dyrektywę group jak następuje:

```
ModuleData      group   Module1Data, Module2Data
```

Jedynym ograniczeniem jest to, że koniec drugiego modułu danych musi być nie większy niż 64 kilobajty od początku pierwszego modułu w pamięci .MASM i linker nie będą łączyć automatycznie tych segmentów i

umiejscowić je razem w pamięci. Jeśli są inne segmenty pomiędzy tymi dwoma w pamięci, wtedy suma wszystkich segmentów musi być mniejsza niż 64kilobajtów.Dal zredukowanie tego problemu, możemy użyć operandu klasy dyrektywy segment. który powie linkerowi aby połączył te dwa segmenty w pamięci poprzez użycie takiej samej nazwy klasy:

```
ModuleData          group        Module1Data, Module2Data
Module1Data         segment      para  public  'MODULES'
                    -
                    -
                    -
Module              ends
                    -
                    -
                    -
Module2Data         segment      byte  public  'MODULES'
                    -
                    -
                    -
Module2Data         ends
```

Z taką deklaracją jak ta powyżej, może użyć „ModuleData" gdziekolwiek MASM pozwoli na nazwę segmentu ,jako operandu instrukcji mov, jako operandu dyrektywy assume, itp. .Poniższy przykład demonstruje zastosowanie nazwy segmentu ModuleData:

```
                    assume       ds:ModuleDAta
ModuleProc          proc         near
                    push         ds               ;zachowanie wartości ds
                    push         ax               ;zachowanie wartości ax
                    mov          ax, ModuleData  ;ładowanie ds segmentem
adress
                    mov          ds, ax           ;ModuleData
                    -
                    -
                    -
                    pop          ax               ;przywrócenie ax i  ds
                    pop          ds
                    ret
ModuleProc          end
                    assume       ds:nothing
```

Oczywiście, używanie dyrektywy group w ten sposób nie możemy poprawić naszego kodu. Faktycznie użycie różnych nazw dla segmentu danych, może kłócić się z zastosowaniem group w ten sposób, właściwie zaciemnia kod. jednakże przypuśćmy ,że mamy sekwencję kodu ,która musi uzyskać dostęp do zmiennych w obu segmentach Module1Data i Module2Data.Jeśli te segmenty były fizycznie i logicznie oddzielone będziemy musieli załadować dwa rejestry segmentowe z adresami tych dwóch segmentów żeby jednocześnie uzyskać do nich dostęp .Będzie to kosztowało nas prefiks przesłonięcia segmentu na wszystkich instrukcjach uzyskujących dostęp do jednego z tych segmentów. Jeśli nie możemy użyczyć dodatkowego rejestru segmentowego, sytuacja będzie nawet gorsza, będziemy musieli stale ładować nową wartość do pojedynczego rejestru segmentowego dla uzyskania dostępu do danych w dwóch segmentach. Możemy uniknąć tego obciążenia poprzez połączenie dwóch logicznych segmentów w pojedynczy fizyczny segment i uzyskiwać dostęp bezpośrednio do grupy zamiast do pojedynczej nazwy segmentu.

Jeśli grupujemy dwa lub więcej segmentów razem, wszystko co robimy to tworzenie pseudo – segmentu który obejmuje segmenty pojawiające się w polu operandu dyrektywy group. Grupowanie segmentów nie zapobiega uzyskiwaniu dostępu do pojedynczych segmentów z pogrupowanej listy .Poniższy kod jest zupełnie poprawny:

```
        assume      ds:Module1Data
        mov         ax, Module1Data
        mov         ds, ax
        -
<Kod, który uzyskuje dostęp do danych w Module1Data>
        -
```

```
assume      ds:Module2Data
mov         ax,Module2Data
mov         ds, ax
-
<Kod, który uzyskuje dostęp dodanych w Module2Data>
-
assume      ds:ModuleData
mov         ax,ModuleData
mov         ds, ax
<Kod, który uzyskuje dostęp do danych w obu – Module1Data i Module2Data>
-
-
-
```

Kiedy assembler tworzy segmenty, zazwyczaj zaczyna od wartości licznika lokacji ustawionej na segment zero. Jednak jeśli grupujemy zbiór segmentów pojawiają się niejasności; zgrupowanie dwóch segmentów powoduje, że MASM i linker łączą zmienne z jednego lub więcej segmentów na końcu pierwszego segmentu z listy. Osiągają to poprzez modyfikowane offsetów wszystkich symboli w łączonych segmentach mimo, że były one symbolami w tym samym segmencie. Dwuznaczność występuje ponieważ MASM pozwala nam odniesienie do symbolu w segmencie lub grupie segmentów .Symbol ma różne offsety w zależności od wybranego segmentu. Rozwiązaniem tej dwuznaczności jest następujący algorytm:

- Jeśli MASM nie wie, że rejestr segmentowy wskazuje na symbol segmentu lub grupę zawierającą segment, MASM wygeneruje błąd.
- Jeśli dyrektywa assume powiązana jest z rejestrem segmentowym nazwy segmentu ale nie powiązana z rejestrem segmentowym grupy nazw, wtedy MASM używa offsetu symbolu wewnątrz segmentu.
- Jeśli dyrektywa powiązana jest z rejestrem segmentowym grupy nazw ale nie powiązana z rejestrem segmentowym nazwy segmentowej symbolu, MASM używa offsetu symbolu grupy.
- Jeśli dyrektywa assume dostarcza rejestru segmentowego powiązanego z oboma symbolami segmentu i grupą, MASM wybierze offset, który nie będzie wymagał prefiksu przesłonięcia segmentu. Na przykład, jeśli dyrektywa assume wyszczególni, że ds wskazuje nazwę grupy a es wskazuje nazwę segmentu, MASM użyje offset grupy jeśli domyślnym rejestrem segmentowym będzie ds ponieważ nie będzie to wymagało aby MASM wyemitował opcod prefiksu przesłonięcia segmentu Jeśli oba wyniki emitują prefiks przesłonięcia segmentu, MASM wybierze offset (i prefiks przesłonięcia offsetu) powiązany z symbolem segmentu.

MASM używa powyższego algorytmu jeśli wyszczególnimy nazwę zmiennej bez prefiksu segmentu. Jeśli wyszczególnimy prefiks przesłonięcia rejestru segmentowego, wtedy MASM może wybrać offset przypadkowy. Często okazuje się być to offset grupy .Poniższa sekwencja instrukcji, bez dyrektywy assume powie MASMowi,że symbol BadOffset w seg1 może tworzyć zły kod wynikowy:

```
DataSegs    group       Data1, Data2, Data3
            -
            -
            -
Data2       segment
            -
            -
            -
BadOffset   word        ?
            -
            -
            -
Data2       ends
            -
            -
            -
            assume      ds:nothing, es:nothing, fs:nothing, gs:nothing
            mov         ax, Data2
despite
            mov         ds, ax
            mov         ax, ds:BadOffset
```

DataSegs
Jeśli chcemy wymusić prawidłowy offset ,używamy nazwy zmiennej zawierającej kompletny adres segment:offset :
;wymusimy użycie offsetu wewnątrz grupy DataSegs uzywając instrukcji takich jak ta:

```
        mov          ax, DataSegs:BadOffset
```

;wymusimy użycie offsetu wewnątrz Data2, używając:

```
        mov          ax, Data2:BadOffset
```

Musimy roztoczyć specjalną troskę kiedy pracujemy z grupą wewnątrz naszego programu asemblerowego. Jeśli zmusimy MASM do użycia offsetu wewnątrz jakiegoś szczególnego segmentu (lub grupy) a rejestr segmentowy nie wskazuje na na ten szczególny segment lub grupę, MASM może nie wygenerować informacji o błędzie a program nie wykona się prawidłowo Odczytując offsety MASM nie pomoże nam znaleźć tego błędu. MASM zawsze wyświetli offset wewnątrz symbolu segmentu w listingu asemblacji. Jedyny rzeczywisty sposób wykrycia, że MASM i linker używają złego offsetu jest zastosowanie debuggera takiego jak CodeView i spojrzenie na aktualne bajty kodu maszynowego stworzonego przez linker i loader.

8.8.12 DLACZEGO ZAWRACMY SOBIE GŁOWĘ SEGMENTAMI?

Po przeczytaniu poprzedniej sekcji, prawdopodobnie zastanawiamy się jakie możliwe są pożytki z zastosowania segmentów w naszych programach. Byłoby doskonale, gdybyśmy zastosowali plik SHELL.ASM jako szkielet dla naszych programów asemblerowych, wtedy możemy mieć łatwiej bez martwienia się o segmenty, grupy, prefiksy przesłonięcia segmentów i pełne nazwy segment:offset. Dla początkujących programistów asemblerowych jest to dobry pomysł, aby zignorować wiele z tego omówienia segmentacji. Jednak istnieją trzy powody dla nauczenia się o segmentacji, jeśli chcemy kontynuować pisanie programów assemblerowych o różnych długościach: ograniczenie segmentu do 64K w trybie rzeczywistym ,modularność programów ,i łączenie z językami wysokiego poziomu.

Kiedy działamy w trybie rzeczywistym, segmenty mogą być długie maksymalnie na 64 kilobajty. Jeśli musimy uzyskać dostęp do więcej niż 64K danych lub kodu w programie, będziemy musieli użyć więcej niż jeden segment. Ten fakt, bardziej niż inne powody, odrzuca programistów od świata segmentacji. Niestety wielu programistów odchodzi od segmentacji. Oni rzadko uczą się wystarczająco o segmentacji, aby pisać programy które uzyskują dostęp do więcej niż 64K danych. W rezultacie ,kiedy wystąpią problemy z segmentacją, ponieważ nie do końca zrozumieli tą koncepcję, winią segmentację za swoje problemy i unikają stosowania segmentacji jak tylko jest to możliwe.

Jest to bardzo złe ponieważ segmentacja jest silnym narzędziem zarządzania pamięcią, które pozwala nam organizować nasze programy w logiczne jednostki (segmenty) które są, w teorii, niezależne od innych. dziedzin inżynierii oprogramowania naucza jak pisać poprawne, duże programy. Modularność i niezależność są dwoma podstawowymi narzędziami inżynierii oprogramowania używanymi do pisania dużych programów które są poprawne i łatwe do pielęgnacji. Rodzina 80x86 dostarcza ,w sprzęcie ,narzędzia do implementacji segmentacji. Na innych procesorach, segmentacja jest wprowadzana wyłącznie programowo .W rezultacie, łatwiej jest pracować z segmentami na procesorach 80x86.

Chociaż ten tekst nie zajmuje się programowaniem w trybie chronionym, warte jest wskazanie, że kiedy działamy w trybie chronionym na procesorach 80286 i późniejszych, sprzęt 80x86 może rzeczywiście zapobiegać aby jeden moduł uzyskiwał dostęp do danych innego modułu (istotnie termin „tryb chroniony” znaczy, że segmenty są chronione przed nieuprawnionym dostępem)Wiele debuggerów dostępnych dla MS-DOS działa w trybie chronionym pozwalając nam na naruszenie obszaru tablicy i segmentu. Soft-ICE i Bounds Checker firmy NuMega są przykładami takich produktów. Większość ludzi którzy pracowali z segmentacją w środowisku trybu chronionego (np. OS/2 czy Windows) ceni sobie zalety oferowane przez segmentację.

Innym powodem dla studiowania segmentacji na 80x86 jest to, że możemy chcieć pisać funkcje asemblerowe które może wywołać program w języku wysokopoziomowy. Ponieważ kompilator HLL czyni pewne założenia o organizacji segmentów w pamięci, będziemy musieli trochę wiedzieć o segmentacji żeby napisać taki kod.

8.9 DYREKTYWA END

Dyrektywa end kończy plik źródłowy języka assemblera. W dodatku mówi MASMowi, że dotarł do końca pliku źródłowego ,operand opcjonalny dyrektywy end mówi MS-DOSowi gdzie ma przekazać sterowanie kiedy program zacznie się wykonywać; to znaczy, wyszczególniamy nazwę procedury głównej jako operand dyrektywy end. Jeśli nie ma operandu dyrektywy end, MS-DOS zacznie wykonywanie począwszy od pierwszego bajtu w pliku .exe. Ponieważ jest to często niepewna gwarancja, że nasz główny program zacznie się od pierwszego bajtu kodu wynikowego w pliku .exe, większość programistów wyszczególnia lokację startową jako operand dyrektywy end. Jeśli używamy pliku SHELL.ASM jako szkieletu dla naszych programów assemblerowych, zauważymy, że dyrektywa end już wyszczególniła procedurę main jako punkt startowy dla programu.

Jeśli nie stogujemy oddzielnej asemblacji i łączymy razem kilka różnych plików kodów wynikowych (zobacz ;"Zarządzanie Dużymi Programami") tylko jeden moduł może mieć program główny. Podobnie ,tylko jeden moduł powinien wyszczególnić lokację startową programu. Jeśli wyszczególnimy więcej niż jedną lokację startową zmylimy linker i wygeneruje on błąd.

8.10 ZMIENNE

Dla zadeklarowania zmiennych globalnych stosujemy pseudo-opcody byte/sbyte/ds,word/sword/dw, dword, sdword/dd, qword/dq i tbyte/dt. Chociaż możemy umieścić nasze zmienne w każdym segmencie (\wliczając w to segment kodu),większość początkujących programistów assemblerowych umieszcza wszystkie swoje zmienne globalne w pojedynczym segmencie danych.

Typowa deklaracja zmiennych przybiera postać:

```
varname         byte            wartość_inicjująca
```

Varname jest nazwą zmiennej którą deklarujemy a wartość_inicjująca jest to wartość ,jaką chcemy aby ta zmienna miała w chwili kiedy program zacznie się wykonywać."?" Jest specjalną wartością inicjującą Oznacza ona, że nie chcemy dać zmiennej wartości inicjującej. Kiedy DOS ładuje program zawierający takie zmienne do pamięci, nie inicjuje tej zmiennej żadną szczególną wartością.

Powyższa deklaracja rezerwuje pamięć dla pojedynczego bajtu. Może to zmienić jakiś inny typ zmiennej poprzez prostą zmianę mnemonika byte na jakiś inny, właściwy pseudo-opcod.

Przeważnie ten tekst będzie zakładał, że deklarujemy wszystkie zmienne w segmencie danych, to znaczy segment na który wskazuje rejestr ds 80x86.W szczególności większość programów umiejscawia wszystkie zmienne w segmencie DSEG (CSEG jest dla kodu, DSEG jest dla danych a SSEG dla stosu).

Ponieważ Rozdział Piąty omawiał deklarację zmiennych, typów danych ,struktur, tablic i wskaźników, więc ten rozdział nie będzie marnował czasu na omawianie tych tematów.

8.11 TYPY ETYKIET

Jedną niezwykła cechą składni assemblera Intela (takiego jak MASM) jest ścisła kontrola typów. Assembler ze ścisłą kontrolą typów kojarzy pewien typ z deklarowanymi symbolami pojawiającymi się w pliku źródłowym i generuje ostrzeżenie lub informację o błędzie jeśli spróbujemy zastosować ten symbol w kontekście na który nie pozwala jego szczególny typ .Chociaż, jest to niezwykłe w assemblerze, większość HLL'i stosuje pewne zasady typowania do deklaracji symboli w pliku źródłowym. Pascal oczywiście jest znany jako język ze ścisłą kontrola typów. Nie możemy w Pascalu przydzielić łańcucha do zmiennej liczbowej lub próbować przydzielić wartość całkowitą do procedury etykiety. Intel, projektując składnię dla assemblera 80x86,zdecydował,że wszystkie przyczyny stosowania języka o ścisłej kontroli typów stosują się również do języka asemblera równie dobrze jak do Pascala. Dlatego też, standardowa składnia assemblera 80x86,np. MASM, narzuca pewne ograniczenia typów na stosowanie symboli wewnątrz programów asemblerowych.

8.11.1 JAK NADAĆ SYMBOL POSZCZEGÓLNYM TYPOM

Symbole, w programie assemblerowym 80x86,mogą być jednym z ośmiu podstawowych typów: byte, word, dword, qword ,tbyte, near, far i abs (stała).zawsze kiedy definiujemy etykietę pseudo-opcodem byte, word, dword, qword lub tbyte,MASM łączy typ tego pseudo-opcodu z etykietą. Na przykład ,następująca deklaracja zmiennej stworzy symbol typu bajt:

```
Bvar            byte            ?
```

Podobnie, definiowanie symbolu dword:

```
DWVar           dword           ?
```

Typy zmiennej nie są ograniczone do podstawowych typów wbudowanych w MASM'a .Jeśli tworzymy własny typ używając dyrektyw typedef lub struct MASM połączy te typy z dołączonymi deklaracjami zmiennych.

Możemy zdefiniować bliskie symbole (znane również jako etykiety instrukcji) na parę różnych sposobów .Po pierwsze, wszystkie deklarowane symbole procedur dyrektywą proc (albo z pustym polem operandu albo near w polu operandu) są bliskimi symbolami. Etykiety instrukcji są również bliskimi symbolami. Etykiety instrukcji przybierają następującą postać:

```
etykieta:       instrukcja
```

Instrukcja, reprezentuje instrukcję 80x86.Zauważmy,że dwukropek musi wystąpić po symbolu. Nie jest to część symbolu, dwukropek informuje assembler, że ten symbol jest etykietą instrukcji i powinien być potraktowany jako typ bliskiego symbolu.

Etykiety instrukcji są często celami instrukcji skoków lub pętli. Na przykład rozważmy następującą sekwencję kodu.:

```
                mov             cx, 25
Loop1:          mov             ax, cx
```

```
                call            PrintInteger
                loop            Loop1
```
Instrukcja loop zmniejsza rejestr cx i przekazuje sterowanie do instrukcji oetykietowanej jako Loop1 dopóki cx nie będzie miało wartości zero.

Wewnątrz procedury ,etykiety instrukcji są lokalne. To znaczy, że zakres etykiet instrukcji wewnątrz procedury jest dostępny tylko dla kodu wewnątrz tej procedury. Jeśli chcemy uczynić symbol globalnym dla procedury, umieścimy dwa dwukropki po nazwie symbolu. W powyższym przykładzie, jeśli musimy odnieść Loop1 na zewnątrz załączonej procedury, powinniśmy użyć kodu:
```
                mov             cx, 25
Loop1::         mov             ax,cx
                call            PrintInteger
                loop            Loop1
```
Ogólnie, dalekie symbole są celami instrukcji skoku i wywołania. Najpowszechniejszą metodą programistyczną użytą do stworzenia dalekiej etykiety jest umieszczenie far w polu operandu dyrektywy proc. Symbole, które są stałymi są normalnie definiowane z dyrektywą equ. Możemy również zadeklarować symbole z różnymi typami używając dyrektyw equ i extm/extem/extemdef. Wyjaśnienie dyrektywy extm pojawi się w sekcji „Zarządzanie Dużymi Programami"

Jeśli zadeklarujemy stałą liczbową używając równania ,MASM przydzieli typ abs (wartość bezwzględna lub stała) do systemu. Dyrektywy równości text i string są dane jako typ text. Możemy również przydzielić dowolnie typ do symbolu używając dyrektywy equ.

8.11.2 WARTOŚCI ETYKIET

Kiedykolwiek zdefiniujemy etykietę używając dyrektywy lub pseudo-opcodu, MASM nada jej typ i wartość. Wartość nadana przez MASM etykiecie jest zazwyczaj wartością bieżącą licznika lokacji. Jeśli zdefiniujemy symbol w dyrektywie równania jako jego operand zazwyczaj wyszczególniamy wartość symbolu. Kiedy napotyka etykietę w polu operandu, jak na przykład w instrukcji loop ,powyżej, MASM podstawia wartość etykiety za tą etykietę.

8.11.3 KONFLIKTY TYPÓW

Ponieważ 80x86 wspiera symbole ścisłej kontroli typów, następne zadane pytanie to:" Po co są one używane? "W dużym skrócie, symbole ścisłej kontroli typów mogą pomóc zweryfikować poprawną operację naszego programu. Rozpatrzmy następującą sekcję kodu:
```
DSEG            segment         public   'DATA'
                -
                -
                -
I               byte            ?
                -
                -
                -
DSEG            ends
CSEG            segment         public 'CODE'
                -
                -
                -
                mov             ax, I
                -
                -
                -
CSEG            ends
                end
```
Instrukcja mov w tym przykładzie próbuje załadować rejestr ax (16 bitów) zmienną o rozmiarze bajta. Teraz mikroprocesor 80x86 jest zupełni zdolny do tej operacji. Załaduje rejestr al z komórki pamięci związanej z I i załaduje rejestr ah z następnej sąsiadującej komórki pamięci (która jest prawdopodobnie najmniej znaczącym bajtem innej zmiennej) Jednakże nie było to pierwotną intencją. Osoba ,która przeczyta ten kod prawdopodobnie zapomni, że I jest zmienną o rozmiarze bajta i założy, że jest to zmienna słowa – co jest zdecydowanym błędem w logice tego programu.

MASM nigdy nie powinien pozwolić instrukcji takiej jak ta powyższa na zasemblowanie bez wygenerowania instrukcji diagnostycznej. Może to pomóc nam zaleźć błąd w programie. Czasami zaawansowani programiści assemblerowi mogą chcieć wykonać instrukcje jak te powyższe. MASM dostarcza

pewnych operatorów sprawdzania zgodności typów, które obchodzą mechanizmy zabezpieczeń MASMa i pozwalają na niepoprawne operacje.

8.12 WYRAŻENIA ADRESOWE

Wyrażenie adresowe jest wyrażeniem algebraicznym, które tworzy wynik liczbowy, które MASM scala do pola przemieszczenia instrukcji. Stała całkowita jest prawdopodobnie najprostszym przykładem wyrażenia adresowego. Assembler po prostu zastępuje wartość stałej liczbowej dla wyszczególnionego operandu. Na przykład, następująca instrukcja wypełnia pole danej bezpośredniej instrukcji mov zerem:

```
        mov         ax, o
```

Inną prostą postacią trybu adresowania jest symbol. Po napotkaniu symbolu, MASM zamienia wartość tego symbolu. Na przykład poniższe dwie instrukcje emitują taki sam kod wynikowy jak instrukcja powyższa:

```
Value   equu    0
        mov     ax, Value
```

Wyrażenia adresowe mogą być bardziej złożone niż te. Możemy użyć różnych arytmetycznych i logicznych operatorów dla modyfikacji podstawowej wartości jakiegoś symbolu lub stałej.

Zapamiętajmy, że MASM oblicza wyrażenie adresowe podczas assemblacji a nie w czasie wykonywania. Na przykład, następująca instrukcja nie załaduje ax z lokacji Var i doda jeden do niej:

```
    mov         ax, Var+1
```

Zamiast tego, instrukcja ta załaduje rejestr al bajtem przechowywanym pod adresem Var1 plus jeden a potem załaduje rejestr ah bajtem przechowywanym pod Var1 plus dwa.

Początkujący programiści często mylą obliczanie robione w czasie assemblacji z tym robionym w trakcie czasu wykonywania. Zapamiętajmy, że MASM oblicza wszystkie wyrażenia adresowe w czasie assemblowania!!!!

8.12.1 TYPY SYMBOLI A TRYBY ADRESOWANIA

Rozważmy następującą instrukcję:

```
        jmp         lokacja
```

W zależności od tego jak etykieta lokacja jest zdefiniowana, ta instrukcja jmp będzie wykonywała jedną z kilu różnych operacji. Jeśli zajrzymy do rozdziału o zbiorze instrukcji 80x86,zauważymy,że instrukcja jmp przybiera kilka form. rekapitulując, oto one:

```
    jmp     label       (krótka)
    jmp     label       (bliska)
    jmp     label       (daleka)
    jmp     reg         (pośredni bliski ,poprzez rejestr)
    jmp     mem/reg     (pośredni bliski, poprzez pamięć)
    jmp     mem/reg     (pośredni daleki, poprzez pamięć)
```

Zauważmy, że MASM używa takiego samego mnemonika (jmp) dla każdej z tych instrukcji; jak on je odróżnia? Sekret leży w operandzie. Jeśli operand jest etykietą instrukcji wewnątrz bieżącego segmentu ,assembler wybiera jedną z pierwszych dwóch postaci w zależności od odległości do instrukcji docelowej. Jeśli operand jest etykietą instrukcji wewnątrz innego segmentu, wtedy assembler wybiera etykietę jmp (daleką).Jeśli operand występujący po instrukcji jmp jest rejestrem, wtedy MASM używa jmp bliskiego pośredniego a program skacze pod adres z rejestru. Jeśli wybrana jest komórka pamięci, assembler używa jednego z następujących skoków:

- NEAR jeśli zmienna była zadeklarowana word/sword/dw
- FAR jeśli zmienna była zadeklarowana dword/sdword/dd

Otrzymamy błędny wynik jeśli użyjemy byte/sbyte/db,qword/dq lub tbyte/dt lub innego typu.

Jeśli wyszczególnimy adres pośredni np. jmp [bx],assembler wygeneruje błąd ponieważ nie może określić czy bx wskazuje na słowo lub podwójne słowo. Po szczegóły jak wyspecyfikować rozmiar, zajrzyj do sekcji o sprawdzaniu zgodności typów w tym rozdziale.

8.12.2 OPERATORY ARYTMETYCZNE I LOGICZNE

MASM rozpoznaje kilka operatorów arytmetycznych i logicznych. Poniższa tablica pokazuje listę takich operatorów:

Operator	Syntax	Description
+	*+expr*	Positive (unary)
-	*-expr*	Negation (unary)
+	*expr + expr*	Addition
-	*expr - expr*	Subtraction
*	*expr * expr*	Multiplication
/	*expr / expr*	Division
MOD	*expr* MOD *expr*	Modulo (remainder)
[ ]	*expr* [*expr*]	Addition (index operator)

Tablica 36 Operatory Arytmetyczne

Operator	Syntax	Description
SHR	expr SHR expr	Shift right
SHL	expr SHL expr	Shift left
NOT	NOT expr	Logical (bit by bit) NOT
AND	expr AND expr	Logical AND
OR	expr OR expr	Logical OR
XOR	expr XOR expr	Logical XOR

Tablica 37 Operatory Logiczne

Operator	Syntax	Description
EQ	*expr* EQ *expr*	True (0FFh) if equal, false (0) otherwise
NE	*expr* NE *expr*	True (0FFh) if not equal, false (0) otherwise
LT	*expr* LT *expr*	True (0FFh) if less, false (0) otherwise
LE	*expr* LE *expr*	True (0FFh) if less or equal, false (0) otherwise
GT	*expr* GT *expr*	True (0FFh) if greater, false (0) otherwise
GE	*expr* GE *expr*	True (0FFh) if greater or equal, false (0) otherwise

Tablica 38 Operatory Porównania

Nie wolno nam pomylić tych operatorów z instrukcjami 80x86!!!Operator dodawania dodaje dwie wartości razem, ich suma staje się operandem instrukcji. To dodawanie jest wykonywane kiedy assemblujemy program a nie podczas jego wykonywania. Jeśli musimy wykonać dodawanie w czasie wykonywania, użyjemy instrukcji add lub adc.

Prawdopodobnie zadamy sobie pytanie "Po co są stosowane te operatory? "Prawda nie jest skomplikowana. Operator dodawania jest używany czasami, odejmowania w mniejszym stopniu, porównania raz na jakiś czas, a reszta nawet mniej. ponieważ dodawanie i odejmowanie są jedynie operatorami stosowanymi w programowaniu w miarę regularnie, to omówienie rozważać będzie tylko te dwa operatory a inne jeśli będzie to wymagane w tym tekście.

Operator dodawania przybiera dwie formy: wyraż + wyraż lub wyraż[wyraż].Na przykład, poniższe instrukcje ładują akumulator ,nie z komórki pamięci COUNT ale z bardzo bliskiej lokacji w pamięci:

```
mov         al, COUNT+1
```

Assembler, po napotkaniu tej instrukcji, obliczy sumę adresu COUNT plus jeden. Wartość wyniku jest adresem pamięci dla tej instrukcji. Instrukcja mov al, pamięć jest trzybajtowa i przybiera postać:
OPCODE | Mniej znaczący bajt przesunięcia | Bardziej znaczący bajt przesunięcia |

Dwa bajty przesunięcia tej instrukcji zawierają sumę COUNT+1.
Forma wyraż[wyraż] operacji dodawania stosowana jest do uzyskania dostępu do elementów tablicy. Jeśli AryData jest symbolem który reprezentuje adres pierwszego elementu tablicy, AryData[5] przedstawia adres piątego bajtu w AryData. Wyrażenie AryData+5 tworzy taki sam wynik, i może być stosowana zamiennie, jednak dla tablic postać wyraż[wyraż] jest trochę bardziej samodokumentująca. Unikniemy pułapki: wyraż1[wyraż2][wyraż3] nie indeksuje (prawidłowo) automatycznie dwu wymiarowej tablicy. Po prostu oblicza sumę wyraż1+wyraż2+wyraż3.

Operator odejmowania pracuje podobnie jak operator dodawania, z wyjątkiem tego, że oblicza różnicę zamiast sumy. Operator ten stanie się ważniejszy kiedy zajmiemy się zmiennymi lokalnymi w Rozdziale 11

Uważajmy, kiedy używamy wielu symboli w wyrażeniach adresowych. MASM ogranicza operacje, które możemy wykonać na symbolach do dodawania i odejmowania i pozwala tylko na następujące formy:

Wyrażenie:	Typ wyniku:
reloc + const	reloc, pod wyszczególniony adres
reloc – const	reloc, pod wyszczególniony adres
reloc – reloc	Stała, której wartość jest liczbą bajtów pomiędzy pierwszym a drugim operandem .Obie zmienne musza fizycznie pojawić się w tym samym segmencie w bieżącym pliku źródłowym.

Reloc oznacza symbol przemieszczalny lub wyrażenie. Może to być nazwa zmiennej, etykieta instrukcji ,nazwa procedury lub każdy inny symbol związany z komórką pamięci w programie. Może to być również wyrażenie tworzące przemieszczalny wynik. MASM nie pozwala na żadną operację inną niż dodawanie lub odejmowanie wyrażeń, której typ wyniku jest przemieszczalny. Na przykład nie możemy my obliczyć iloczynu dwóch przemieszczalnych symboli.

Pierwsze dwie z powyższych form są bardzo popularne w programach assemblerowych .takie wyrażenia adresowe często składają się z pojedynczego przemieszczalnego symbolu i pojedynczej stałej (np. var+1).nie będziemy mogli użyć trzeciej formy bardzo często ,ale jest bardzo użyteczna raz na jakiś czas. Możemy użyć tej formy wyrażenia adresowego do obliczenia odległości, w bajtach między dwoma punktami w naszym programie. Symbol procsize w następującym kodzie oblicza rozmiar Proc1:

```
Proc1       proc      near
            push      ax
            push      bx
            push      cx
            mov       cx, 10
            lea       bx, SomeArray
            mov       ax, 0
ClrArray    mov       [bx],ax
            add       bx,2
            loop      ClrArray
            pop       cx
            pop       bx
            pop       ax
            ret
Proc1       endp

Procsize    =         $ - Proc1
```

„$" jest specjalnym symbolem, którego MASM używa do określania bieżącego offsetu wewnątrz segmentu (np. licznik lokacji)jest to symbol przemieszczalny ,więc powyższe równanie oblicza długość procedury Proc1,w bajtach.

Operand operatora innego niż dodawanie lub odejmowanie musi być stała lub wyrażeniem dającym stałą (np. „$-Proc1" daje stałą wartość).Głównie używamy tych operatorów w makrach i z dyrektywami asemblacji warunkowej.

8.12.3 KOERCJA

Rozważmy następujący segment programu:

```
DSEG        segment   public 'DATA'
I           byte      ?
```

```
J               byte            ?
DSEG            ends
CSEG            segment
                -
                -
                -
                mov             al, I
                mov             ah, J
                -
                -
                -
CSEG            ends
```

Ponieważ I i J są przyległe ,nie musimy używać dwóch instrukcji mov do załadowania al i ah, prosta instrukcja mov ax, I zrobiłaby to samo. Niestety, assembler będzie się wzdragał przed mov ax ,I ponieważ I jest bajtem. Assembler będzie narzekał jeśli spróbujemy potraktować to jako słowo.. Jak zobaczymy, prawdopodobnie będzie kilka sytuacji kiedy potraktujemy zmienną bajtową jako słowo (lub słowo jako bajt lub podwójne słowo, lub potraktować podwójne słowo jako coś innego).

Czasowa zmiana typu etykiety dla jakiegoś szczególnego wystąpienia to koercja. Wyrażenie może być sprowadzone do innego typu przez zastosowanie operatora ptr. Użyjemy operatora ptr jak następuje:

```
                type PTR wyrażenie
```

Type jest jednym z typów byte ,word ,dword, tbyte ,near ,far lub innego a wyrażenie jest ogólnym wyrażeniem, które jest adresem jakiegoś obiektu .Operator koercji zwraca wyrażenie o takiej samej wartości jak wyrażenie, ale z typem wyszczególnionym przez type. Rozwiązaniem powyższego problemu będzie użycie instrukcji języka assemblera:

```
                mov             ax, word ptr I
```

Informuje to assembler, żeby wyemitował kod, który załaduje rejestr ax słowem spod adresu I .Oczywiście załaduje al I a ah J.

Kod który używa wartości podwójnego słowa często robi się rozległy poprzez zastosowanie operacji koercji .Ponieważ lds i lea są jedynymi 32 bitowymi instrukcjami na pre –procesorach 80386,nie możemy (bez koercji) przechować wartości całkowitych w 32 bitowych zmiennych używając instrukcji mov na tych wcześniejszych CPU. jeśli zadeklarujemy DBL używając pseudo-opcodu dword, wtedy instrukcja w postaci mov DBL, ax wygeneruje błąd ponieważ próbujemy przenieść 16 bitową wielkość do 32 bitowej zmiennej. Przechowywanie wartości w zmiennej podwójnego słowa wymaga użycia operatora ptr. Poniższy kod demonstruje jak przechowywać rejestry ds i bx w zmiennej podwójnego słowa DBL:

```
                mov             word ptr DBL, bx
                mov             word ptr DBL+2, ds
```

Będziemy używać tej techniki często przy Standardowej Bibliotece UCR i wywołaniach MS-DOS zwracających wartość podwójnego słowa w parze rejestrów.

Ostrzeżenie: Jeśli wykonujemy koercje instrukcji jmp wymagającą skoku far do etykiety near, (far jmp dłużej się wykonuje),nasz program będzie pracował dobrze. Jeśli wykonamy koercję call wymagającą dalekiego wywołania do bliskiego podprogramu, kierujemy się prosto do kłopotów. Pamiętajmy, że dalekie wywołanie odkłada rejestr cs na stos (z adresem powrotnym)Kiedy wykonamy bliską instrukcję ret, stara wartość cs nie będzie ściągnięta ze stosu ,pozostawiając śmieci na stosie Bardzo blisko położone instrukcje pop i ret nie działają poprawnie ponieważ zdejmują wartość cs ze stosu zamiast oryginalną wartość odłożoną na stos.

Wyrażenie poddane koercji może przydać się czasami. Innym razem jest niezbędne. Jednakże nie powinniśmy unikać koercji ponieważ sprawdzanie typu danych jest silnym narzędziem wbudowanym w MASM. Poprzez użycie koercji, możemy zlekceważyć to zabezpieczenie dostarczane przez assembler. Dlatego też, zawsze uważajmy kiedy przesłaniamy typ symbolu operatorem ptr.

Jedyne miejsce gdzie potrzebujemy koercji jest instrukcja mov pamięć, dana bezpośrednia. Rozważmy następującą instrukcję:

```
                mov             [bx], 5
```

Niestety, assembler nie ma sposobu aby dowiedzieć się czy bx wskazuje pozycję bajtu, słowa lub podwójnego słowa w pamięci. Wartość operandu bezpośredniego nie jest używana .Pomimo, że pięć jest wielkością bajtu ,instrukcja ta może przechowywać wartość 0005 w zmiennej word, lub 00000005 w zmiennej podwójnego słowa. Jeśli spróbujemy zasemblować tę instrukcję, assembler wygeneruje błąd ,w skutek tego, że musimy wyszczególnić rozmiar operandu pamięci. Możemy łatwo to osiągnąć używając operatorów byte ptr, word ptr i dword ptr jak następuje:

```
                mov             byte ptr [bx],5         ;dla zmiennej bajtowej
                mov             word ptr [bx],5         ;dla zmiennej słowa
                mov             dword ptr [bx], 5       ;dla zmiennej podwójnego słowa
```

Leniwy programista może narzekać, że pisanie łańcuchów jak „word ptr" lub „far ptr" wymaga zbyt dużo pracy. Czyż nie byłoby milej ,gdyby Intel wybrał pojedynczego symbolu znaku zamiast tych długich fraz? Cóż, przestańmy narzekać i pamiętajmy o dyrektywie textequ. Z dyrektywami równania możemy zastąpić długi łańcuch taki jak „word ptr" na krótszy symbol. Znajdziemy takie dyrektywy równania w wielu programach:

```
byp       textequ  <byte ptr>      ;pamiętaj ,"bp" jest zarezerwowanym symbolem
wp        textequ <word ptr>
dp        textequ <dword ptr>
np.       textequ <near ptr>
fp        textequ <far ptr>
```

Z dyrektywami równań, jak powyższe, możemy użyć instrukcji jak pokazano poniżej:

```
          mov     byp [bx], 5
          mov     ax, wp I
          mov     wp DBL, bx
          mov     wp DBL+2, ds
```

8.12.4 TYPY OPERATORÓW

Operator koercji „xxxx ptr" jest przykładem operatora. Wyrażenie MASM posiadają dwa główne atrybuty: wartość i typ. Operatory arytmetyczne ,logiczne i relacyjne zmieniają wartość wyrażenia. Operatory typów zmieniają ich typ. Poprzednia sekcja demonstrowała jak operator ptr może zmienić typ wyrażenia. Jest kilka dodatkowych operatorów typów.

Operator	Syntax	Description
PTR	byte ptr *expr* word ptr *expr* dword ptr *expr* qword ptr *expr* tbyte ptr *expr* near ptr *expr* far ptr *expr*	Coerce *expr* to point at a byte. Coerce *expr* to point at a word. Coerce *expr* to point at a dword. Coerce *expr* to point at a qword. Coerce *expr* to point at a tbyte. Coerce *expr* to a near value. Coerce *expr* to a far value.
short	short *expr*	*expr* must be within ±128 bytes of the current jmp instruction (typically a JMP instruction). This operator forces the JMP instruction to be two bytes long (if possible).
this	this *type*	Returns an expression of the specified type whose value is the current location counter.
seg	seg *label*	Returns the segment address portion of *label*.
offset	offset *label*	Returns the offset address portion of *label*.
.type	type *label*	Returns a byte that indicates whether this symbol is a variable, statement label, or structure name. Superceded by opattr.
opattr	opattr *label*	Returns a 16 bit value that gives information about *label*.
length	length *variable*	Returns the number of array elements for a single dimension array. If a multi-dimension array, this operator returns the number of elements for the first dimension.
lengthof	lengthof *variable*	Returns the number of items in array *variable*.
type	type *symbol*	Returns a expression whose type is the same as *symbol* and whose value is the size, in bytes, for the specified symbol.
size	size *variable*	Returns the number of bytes allocated for single dimension array *variable*. Useless for multi-dimension arrays. Superceded by sizeof.
sizeof	sizeof *variable*	Returns the size, in bytes, of array *variable*.
low	low *expr*	Returns the L.O. byte of *expr*.
lowword	lowword *expr*	Returns the L.O. word of *expr*.
high	high *expr*	Returns the H.O. byte of *expr*.
highword	highword *expr*	Returns the H.O. word of *expr*.

Tablica 39: Operatory typu

Operator short pracuje wyłącznie z instrukcją jmp .Pamiętamy, że są dwie instrukcje jmp bliskie bezpośrednie, jedna która ma zakres 128 bajtów, druga ma 32,768 bajtów. MASM automatycznie generuje krótki skok jeśli adres docelowy jest większy niż 128 bajtów przed bieżącą instrukcją. Operator ten jest głównie obecny ze względu na kompatybilność ze starszymi wersjami MASMa.

Operatora this tworzy wyrażenie z wyszczególnionym typem którego wartość jest bieżącym licznikiem lokacji. Na przykład instrukcja mov bx, this word, załaduje rejestr bx wartością 8B1Eh,opcodem dla mov bx, pamięć. Adres this word jest adresem opcodu dla tej właśnie instrukcji!. Operatora this używamy głównie z dyrektywą equ dającą symbol typu innego niż stała .Na przykład, rozważmy następującą instrukcję:

```
HERE        equ         this near
```

Ta instrukcja przydziela bieżąca wartość licznika lokacji do HERE i ustawia typ HERE na near, Oczywiście mogłoby to być zrobione dużo łatwiej poprzez umiejscowienie etykiety HERE: w tej samej linii **.**Rozważmy coś takiego:

```
Warray      equ     this word
Barray      byte    200 dup  (?)
```

W tym przykładzie symbol Barray jest typu byte. Dlatego też, instrukcje uzyskujące dostęp do Barray musi zawierać operand byte .MASM sygnalizuje instrukcję mov ax,BArray+8 jako błąd .Jednak zastosowanie symbolu WArray pozwala nam uzyskać dostęp dokładnie do tej samej komórki pamięci (ponieważ Warray ma wartość licznika lokacji bezpośrednio przed napotkaniem pseudo-opcodu byte) więc mov ax,WArray+8 uzyskuje dostęp do BArray+8.Zauważmy,że następujące dwie instrukcje są podobne:

```
            mov         ax, word ptr BArray+8
            mov         ax, WArray+8
```

Operator seg robi dwie rzeczy. Po pierwsze wydziela część segmentową wyszczególnionego adresu, po drugie konwertuje typ wyszczególnionego wyrażenia z adresu do stałej. Instrukcja w postaci mov ax, ser symbol zawsze ładuje akumulator stałą odpowiadającą adresowi części segmentu symbol. Jeśli symbol jest nazwą segmentu. MASM automatycznie zastępuje adres paragrafu segmentu dla nazwy. Jednakże, jest całkiem poprawne zastosowanie operatora seg również. Poniższe dwie instrukcje są identyczne jeśli dseg jest nazwą segmentu:

```
            mov     ax, dseg
            mov     ax, seg dseg
```

Offset pracuje podobnie jak seg, z tym, że zwraca offset części wyszczególnionego wyrażenia zamiast część segmentu. jeśli VAR1 jest zmienną słowa, mov ax ,VAR1 zawsze załaduje dwa bajty spod adresu wyszczególnionego przez VAR1 do rejestru ax. Instrukcja mov ax, offset VAR1 ładuje offset (adres) VAR1 do rejestru ax. Zauważmy, że możemy użyć instrukcji lea lub instrukcji mov z operatorem offset do załadowania adresu zmiennej skalarnej do rejestru 16 bitowego. Poniższe dwie instrukcje, obie ładują bx adresem zmiennej J;

```
            mov         bx, offset J
            mov         bx, J
```

Instrukcja lea jest bardziej elastyczna ponieważ możemy wyszczególnić każdy tryb adresowania pamięci, operator offset pozwala nam na pojedynczy symbol (tj. tryb adresowania „tylko przemieszczenie")Większość programistów używa formy mov dla zmiennych skalarnych a instrukcji lea dla innych trybów adresowania. Jest tak ponieważ instrukcja mov będzie szybsza na wcześniejszych procesorach.

Jednym bardzo popularnym zastosowaniem operatorów seg i offset jest inicjacja rejestru segmentu i wskaźnika adresem segmentowym jakiegoś obiektu. Na przykład ładując es:di adresem SomeVar, możemy użyć poniższego kodu:

```
            mov         di, seg SomeVar
            mov         es, di
            mov         di, offset SomeVar
```

Ponieważ nie możemy załadować stałej bezpośrednio do rejestru segmentowego, powyższy kod kopiuje część segmentową adresu do di a potem kopiuje di do es przed skopiowaniem offsetu do di. Kod ten używa rejestru di do kopiowania części segmentowej adresu do es, więc nie będzie wpływał na kilka innych rejestrów

Opattr zwraca 16 bitową wartość dostarczając określonej informacji o wyrażeniu występującym po nim. Operator .type jest starsza wersją opattr, która zwraca mniej znaczące osiem bitów tej wartości. Każdy bit wartości tego operatora ma następujące znaczenie:

Bit(s)	Meaning
0	References a label in the code segment if set.
1	References a memory variable or relocatable data object if set.
2	Is an immediate (absolute/constant) value if set.
3	Uses direct memory addressing if set.
4	Is a register name, if set.
5	References no undefined symbols and there is no error, if set.
6	Is an SS: relative reference, if set.
7	References an external name.
8-10	000 - no language type 001 - C/C++ language type 010 - SYSCALL language type 011 - STDCALL language type 100 - Pascal language type 101 - FORTRAN language type 110 - BASIC language type

Tablica 40: Zwracane wartości OPATTR/.TYPE

Bity języka są dla programistów, którzy piszą kody dla języków wysokopoziomowych takich jak Pascal czy C++. Takie programy używają uproszczonych dyrektyw segmentowych i cech HLLi MASMa.

Będziemy mogli normalnie używać tych wartości z dyrektywami asemblacji warunkowej MASMa i makrami. Pozwoli to nam wygenerować różne sekwencje instrukcji w zależności od typu parametrów makra lub bieżącej konfiguracji assemblacji.

Operatory size ,sizeof, length i lengthof obliczają rozmiar zmiennych (wliczając w to tablice) i zwracają te rozmiary i ich wartości. Zwykle nie powinniśmy używać size i length. Operatory sizeof i lengthof zastąpiły te operatory. Size i length nie zawsze zwracają sensowne wartości dla różnych operandów. MASM 6.x zawiera je tylko dla kompatybilności ze starszymi wersjami assemblera. Jednakże, później w rozdziale zobaczymy przykład gdzie używamy tych operatorów. Operator sizeof zmiennej zwraca liczbę bajtów bezpośrednio alokowanych w wyszczególnionej zmiennej. Ilustruje to poniższy przykład:

```
a1      byte    ?                       ;sizeof (a1)=1
a2      word    ?                       ;sizeof(a2)=2
a4      dword   ?                       ;sizeof)(a4)=4
a8      real8   ?                       ;sizeof(a8)=8
ary0    byte    10 dup (0)              ;sizeof (ary0) = 10
ary1    word    10 dup(10 dup (0)       ;sizeof(ary1) = 200
```

Możemy również użyć operatora sizeof do obliczenia rozmiaru, w bajtach, struktury innych typów danych .Jest to bardzo użyteczne dla obliczania indeksów w tablicy używając formuły z Rozdziału Czwartego:

Adres_Elelmntu := adres_bazowy+indeks*Rozmiar_Elementu

Możemy otrzymać rozmiar elementu tablicy lub struktury używając operatora sizeof. Więc jeśli mamy tablicę struktury ,możemy obliczyć indeks do tablicy jak następuje:

```
        .286                            ;dozwolone instrukcje 80286
s       struct
        <pewna liczba pól>
s       ends
        -
        -
        -

array   s       16 dup ({})             ;tablica 16 elementów „s”
        -
        -
        -
        imul    bx, I, sizeof s         ;obliczenie BX :=I * rozmiar elementu
```

```
        mov     al, array[bx].nazwa pola
```

Możemy również zastosować operator sizeof do innych typów danych aby uzyskać ich rozmiary w bajtach .Na przykład ,sizeof byte zwraca 1, sizeof word zwraca dwa a sizeof dword zwraca 4.Oczywiście zastosowanie tego operatora dla wbudowanych w MASM typów danych jest wątpliwe ponieważ rozmiar tych obiektów jest stały. Jednakże, jeśli stworzymy swój własny typ danych używając typedef, nabierze sens obliczanie rozmiaru tego obiektu przez użycie operatora sizeof:

```
Integer         typedef  word
Array           integer  16  dup  (?)
                -
                -
                -
                imul     bx, bx, sizeof integer
                -
                -
                -
```

W powyższym kodzie, sizeof integer będzie zwracał dwa podobnie jak sizeof word. Jednak jeśli zmienimy instrukcję typedef tak aby integer wskazywał dword zamiast word, operand sizeof integer automatycznie zmieni jego wartość na cztery ,odzwierciedlając nowy rozmiar integer.

Operator lengthof zwraca całkowitą liczbę elementów w tablicy. Dla powyższej zmiennej Array,lengthof Array zwróci 16.jeśli mamy dwuwymiarową tablicę, lengthof zwróci całkowitą liczbę elementów w tej tablicy.

Kiedy użyjemy operatorów lengthof i sizeof z tablicami, musimy zapamiętać, że jest możliwe zadeklarowanie tablic w sposób w który MASM może źle zinterpretować. Na przykład, poniższe instrukcje deklarują tablice zawierające osiem słów:

```
A1              word            8 dup (?)
A2              word            1,2,3,4,5,6,7,8
;Notka: „\" jest symbolem „kontynuowania linii" Mówi MASMowi aby dołączył następną linie do końca bieżącej ;linii
A3              word            1,2,3,4 \
                                5,6,76,8
A4              word            1,2,3,4
                word            5,6,7,8
```

Zastosowanie operatorów sizeof i lengthof dla A1,A2 i A3 daje 16 (sizeof) i 8 (lengthof).Jednakże sizeof(A4) daje osiem a lengthof(A4) daje cztery. Dzieje się tak ponieważ MASM sądzi, że tablice zaczynają się i kończą pojedynczymi deklaracjami danych .Chociaż A4 deklaruje rezerwację w pamięci osiem kolejnych słów, podobnie jak trzy inne powyższe deklaracje, MASM sądzi, że dwie dyrektywy word deklarują dwie oddzielne tablice zamiast tablicy pojedynczej. Więc jeśli chcemy inicjować elementy dużej tablicy lub tablicy wielowymiarowej i również chcemy móc zastosować operatory lengthof i sizeof do tej tablicy ,powinniśmy użyć postaci A3 dla deklaracji zamiast A4

Operator type zwraca stałą, która jest liczbą bajtów wyszczególnionego operandu. Na przykład, type(word) zwraca wartość dwa Ta rewelacja nie jest szczególnie interesująca ponieważ operatory size i sizeof również zwracają tą wartość jednak, kiedy używamy operatora type do porównania operatorów (np,. ne,le,lt,gt i ge) porównanie to daje wynik prawdziwy tylko jeśli typy operandów są takie same. rozważmy następujące definicje:

```
Integer         typedef         word
J               word            ?
K               sword           ?
L               integer         ?
M.              word            ?
                byte            type  (J)  eq word              ;wartość = 0FFh
                byte            type  (J)  eq  sword            ;wartość = 0
                byte            type  (J)  eq  type (L)         ;wartość = 0FFh
                byte            type  (J)  eq  type (M)         ;wartość = 0FFh
                byte            type  (L)  eq integer           ;wartość = 0FFh
                byte            type (K)   eq  dword            ;wartość = 0
```

Ponieważ powyższy kod zmienia integer na word, MASM traktuje wartości całkowite i słowa jako ten sam typ. Zauważmy, że z wyjątkiem ostatniego przykładu, wartość po obu stronach operatora eq to dwa. Dlatego też, kiedy używamy operatorów porównania z operatorem type MASM porównuje więcej niż tą wartość .Dlatego też, type i sizeof nie są synonimami. Np.

```
                byte            type (J) eq type (K)            ;wartość = 0
```

	byte	(sizeof J) equ (sizeof K)	;wartość = 0FFh

Operator type jest zwłaszcza użyteczny kiedy używamy warunkowych dyrektyw assemblacji MASMa. Powyższe przykłady demonstrują również inną interesującą cechę MASMa. Jeśli użyjemy typu name wewnątrz wyrażenia, MASM potraktuje go jak gdybyśmy wprowadzili „typ(name)" gdzie name jest symbolem danego typu .W szczególności, wyszczególniona nazwa typu zwraca rozmiar, w bajtach, obiektu danego typu. Rozważmy następujący przykład:

```
Integer     typedef     word
s           struct
d           dword       ?
w           word        ?
b           byte        ?
s           ends
            byte        word                ;wartość = 2
            byte        sword               ,wartość = 2
            byte        byte                ;wartość = 1
            byte        dword               ;wartość = 4
            byte        s                   ;wartość = 7
            byte        word  eq  word      ;wartość = 0FFh
            byte        word  eq  sword     ;wartość = 0
            byte        b  eq  dword        ;wartość = 0
            byte        s  eq  byte         ;wartość = 0
            byte        word  eq Integer    ;wartość = 0FFh
```

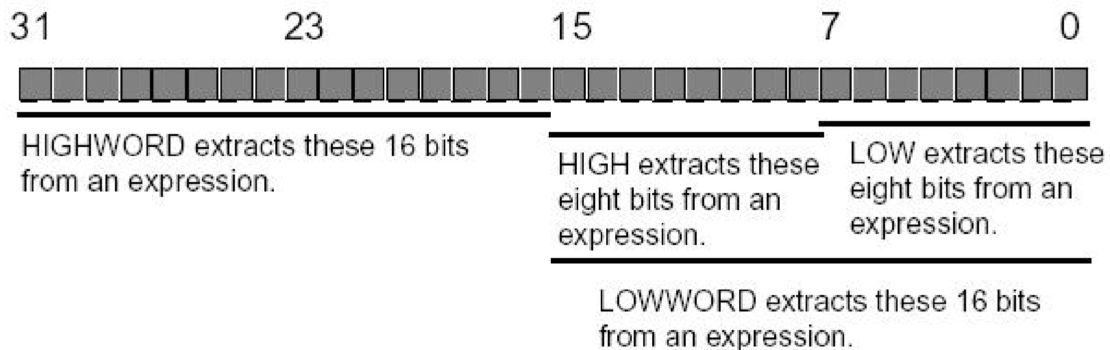
Operatory high i low, podobnie jak offset i seg zmieniają typ wyrażenia z jakiejkolwiek na stałą. Operatory te rózwniż wpływają na wartość – rozkładają go na bardziej i mniej znaczące bajty. Operator high ekstrahuje bity wyrażenia od osiem do piętnaście, operator low ekstrahuje i zwraca bity od zera do siedem. Highword i lowword ekstrahuje bardziej i mniej znaczące 16 bitów wyrażenia zobacz rysunek 8.7)

Możemy wyciągnąć bity 16 – 23 i 24-31 używając wyrażeń w postaci low (highword (expr)) i high(highword(expr)),odpowiednio.

8.12.5 OPERATOR PIERSZEŃSTWA

Chociaż będziemy rzadko musieli używać złożonych wyrażeń adresowych stosując dwa lub więcej niż dwa operandy i pojedynczy operator, czasami potrzebujemy je zastosować .MASM wspiera pojedynczy operator konwencji pierwszeństwa oparty na następujących zasadach:

- MASM wykonuje operatory od najwyższego priorytetu



Rysunek 8.7:Operatory HIGHWORD,LOWWORD,HIGH i LOW

- Operatory równania są lewo łączne i obliczane od lewej do prawej
- Nawiasy unieważniają normalne pierwszeństwa

Precedence	Operators
(Highest)	
1	length, lengthof, size, sizeof, ( ), [ ], < >
2	. (structure field name operator)
3	CS: DS: ES: FS: GS: SS: (Segment override prefixes)
4	ptr offset set type opattr this
5	high, low, highword, lowword
6	+ - (unary)
7	* / mod shl shr
8	+ - (binary)
9	eq ne lt le gt ge
10	not
11	and
12	or xor
13	short .type
(Lowest)	

Tablica 41: Operatory pierwszeństwa

Nawiasy powinny tylko otaczać wyrażenia. Operatory takie jak sizeof i lengthof wymagaj nazwy typu, nie wyrażeń .Nie pozwalają nam otoczyć nawisami nazw. Dlatego też „sizeof X" jest poprawne, ale „sizeof(X)" nie .Zapamiętajmy kiedy używamy nawiasów do unieważnienia operatorów pierwszeństwa w wyrażeniu. Jeśli MASM generuje błąd, możemy będziemy musieli przestawić w naszym wyrażeniu.

Podobnie jak dla wyrażeń w językach wysokiego poziomu, dobrym pomysłem jest używanie zawsze nawiasów okrągłych do wyraźnego określenia stanu pierwszeństwa we wszystkich złożonych wyrażeniach adresowych (złożony w znaczeniu ,że wyrażenie ma więcej niż jeden operator).Uczyni to wyrażenia bardziej czytelnymi i pozwoli uniknąć błędów pierwszeństwa.

8.13 ASSEMBLACJA WARUNKOWA

MASM dostarcza bardzo silnych udogodnień assemblacji warunkowej. Z asemblacją warunkową, możemy zadecydować ,w oparciu o pewne warunki, czy MASM będzie assemblował kod. Jest kilka dyrektyw assemblacji warunkowej, poniższa sekcja omówi większość z nich.

Ważne jest aby uświadomić sobie, że dyrektywy te wyliczają wyrażenia w czasie assemblacji a nie w czasie wykonania .Dyrektywa assemblacji warunkowej nie jest tym samym co instrukcja „if" Pascala lub C. Jeśli jesteśmy zaznajomieni z C, dyrektywa #ifdef w C jest pobieżnym odpowiednikiem dyrektyw assemblacji warunkowej MASMa.

Dyrektywy assemblacji warunkowej MASMa są ważne ponieważ pozwalają nam generować różne kody wynikowe dla różnych operacji środowiska i różnych sytuacji. Na przykład przypuśćmy, że chcemy napisać program, który będzie pracował na wszystkich maszynach, ale chcielibyśmy zoptymalizować kod dla 80386 i późniejszych procesorów. Oczywiście nie możemy wykonać kodu 80386 na procesorze 8086,więc jak możemy rozwiązać ten problem?

Jednym z możliwych rozwiązań jest określenie typu procesora w czasie wykonania i wykonanie różnych sekcji kodu w programie w zależności od obecności lub absencji 386 lub późniejszych CPU. Problem w takim podejściu jest taki, że nasz program musi zawierać dwie sekwencje kodu – optymalna sekwencję dla 80386 i kompatybilną sekwencję dla 8086.Na każdym danym systemie, CPU wykona tylko jedną z tych sekwencji w programie, więc druga sekwencja będzie marnowała pamięć i może mieć niekorzystny wpływa na cache w systemie.

Drugą możliwością jest napisanie dwóch wersji kodu, jeden, który używa tylko instrukcji 8086 i jeden, który stosuje pełen zbiór instrukcji. Podczas instalacji, użytkownik (lub program instalacyjne) wybiera wersję 80386 jeśli mamy procesor 80386 lub późniejszy. W przeciwnym wypadku wybiera wersję 8086.Jest to nieznaczne zwiększenie kosztów programu, ponieważ wymaga to więcej przestrzeni na dysku, program będzie zużywał mniej pamięci podczas działania. Problem jaki napotkamy to to, że będziemy musieli utrzymywać dwie

odrębne wersje programu. Jeśli poprawimy błąd w wersji kodu dla 8086,prawdopodbnie będziemy musieli poprawić ten sam błąd w programie dla 80386.utrzeymywanie wielu plików źródłowych jest trudnym zadaniem.

Trzecim rozwiązaniem jest zastosowanie asemblacji warunkowej. Z asemblacją warunkową możemy połączyć wersje kodu dla 8086 i 80386 w tym samym pliku źródłowym. Podczas asemblacji możemy warunkowo wybierać czy MASM assembluje wersję kodu dla 8086 czy 80386.Poprzez dwukrotną asemblacje kodu ,możemy stworzyć wersje kodu dla 8086 i 80386.Ponieważ obie wersje kodu pojawiają się w tym samym pliku źródłowym, program będzie dużo łatwiejszy do utrzymania ponieważ nie będziemy musieli poprawiać tych samych błędów w dwóch oddzielnych plikach źródłowych. Musimy poprawić ten sam błąd dwa razy w dwóch oddzielnych sekwencjach kodu w programie, ale generalnie błąd pojawi się w dwóch sąsiadujących sekwencjach kodu, więc jest mniej prawdopodobne, że zapomnimy dokonać zmian w obu miejscach.

Dyrektywy asemblacji warunkowej MASMa są szczególnie użyteczne wewnątrz makr. Mogą one pomóc nam stworzyć sprawnie działający kod kiedy makro normalnie tworzy sub -optymalny kod .Po więcej informacji o makrach i jak używać asemblacji warunkowej wewnątrz makr zajrzyj do „Makra".

Makra i asemblacja warunkowa właściwie dostarczają „języka programowania wewnątrz języka programowania" .Makra i asemblacja warunkowa pozwalają nam pisać programy (w „języku makra") które tworzą część kodu assemblera. Wprowadza to niezależny sposób generowania błędów w naszych aplikacjach. Nie tylko można tworzyć błędy w naszym kodzie assemblerowym, ale możemy również wprowadzać błędy w naszym kodzie makra (np. asemblacja warunkowa),co kończy się tworzeniem błędów w kodzie asemblerowym. Zapamiętajmy ,że jeśli dostajemy kod zbyt wyrafinowany kiedy u używamy asemblacji warunkowej ,tworzymy programy, które są zbyt trudne do odczytania ,zrozumienia i zdebuggowania.

8.13.1 DYREKTYWA IF

Dyrektywa if używa następującej składni:

```
        if      wyrażenie
<sekwencja instrukcji>
        else    ;to jest opcjonalne
<sekwencja instrukcji>
        endif
```

MASM oblicza wyrażenie. Jeśli nie jest to wartość zerowa, wtedy MASM zasembluje instrukcje pomiędzy dyrektywami if i else (lub endif, jeśli else nie jest obecna).Jeśli wyliczone wyrażenie jest zerem (fałsz) a sekcja else występuje, MASM zasembluje instrukcje pomiędzy dyrektywami else a endif. Jeśli else nie występuje, a wartość wyrażenia jest fałszem, wtedy MASM nie zasembluje żadnego kodu pomiędzy dyrektywami if i endif.

Ważną rzeczą do zapamiętania jest to ,że wyrażenie ma być wyrażeniem, które MASM może wyliczyć w czasie asemblacji. To znaczy musi obliczyć stałą. Stałą jawna i wartość, które tworzą typ operatorów MASMa są dostępne w wyrażeniach dyrektywy if. Na przykład przypuśćmy ,że chcemy zasemblować kod dla dwóch różnych procesorów jak opisano powyżej. Możemy użyć instrukcji podobnych do tych:

```
Procesor    =       80386           ;ustawia 8086 dla kodu tylko-8086
            -
            -
            -
            if      Procesor eq 80386
            shl     ax, 4
            else                    ,musi być procesor 8086
            mov     cl, 4
            shl     ax, cl
            endif
```

Są inne sposoby osiągnięcia tej samej rzeczy. MASM dostarcza zmiennych wbudowanych które mówią nam czy assemblujemy kod dla wyszczególnionego procesora. Ale o tym później.

8.13.2 DYREKTYWA IFE

Dyrektywa ife jest używana dokładnie jak dyrektywa if, z wyjątkiem tego, że assembluje kod po dyrektywie ife tylko jeśli wyliczona wartość wyrażenia to zero (fałsz),zamiast prawda (nie – zero)

8.13.3 IFDEF I IFNDEF

Te dwie dyrektywy wymagają pojedynczego symbolu jako operandu. Ifdef będzie assemblować powiązany kod jeśli symbol jest zdefiniowany. Ifndef będzie assemblował powiązany kod jeśli symbol nie jest zdefiniowany. Użycie else i endif kończy sekwencję assemblacji warunkowej.

Dyrektywy te są zwłaszcza popularne dla zawartego lub nie zawartego kodu w programie asemblerowym operującym w pewnych specjalnych przypadkach. Na przykład, możemy użyć instrukcji takich jak poniższe zawierające instrukcje debuggujące w naszym kodzie:

```
        ifdef       DEBUG
        <tu miejsce na instrukcje debuggujące>
        endif
```

Aby uruchomić kod debuggujący po prostu definiujemy symbol DEBUG gdzieś na początku naszego programu (przed pierwszym odwołaniem ifdef do DEBUG).Automatyczne wyeliminowanie kodu debuggującego polega po prostu na skasowaniu definicji DEBUG .Możemy zdefiniować DEBUG używając prostej instrukcji:

```
DEBUG       =       0
```

Zauważmy, że wartość przydzielona do DEBUG jest nieistotna. Tylko fakt, że mamy (lub nie mamy) zdefiniowanego tego symbolu jest ważny.

8.13.4 IFB,IFNB

Dyrektywy te, są użyteczne głównie w makrach do sprawdzania czy operand jest pusty (ifb) lub nie pusty (ifnb).Rozważmy następujący kod:

```
Blank           textequ     <>
NotBlank        textequ     <not blank>

                ifb         Blank
                <ten kod się assembluje>
                endif
                ifb         NotBlank
                <ten kod nie>
                endif
```

Ifnb pracuje przeciwnie do ifb. To znaczy, assemblowałby instrukcje powyższe, których nie assembluje ifb i vice versa.

8.13.5 IFIDN,IFDIF,IFIDNI I IFDIFI

Te dyrektyw asemblacji warunkowej pobierają dwa operandy i przetwarzają powiązany kod jeśli operandy są identyczne (ifidn),różne (ifdif),identyczne bez rozróżniania liter(ifidni) lub różne bez rozróżniania liter. Ich składnia:

ifidn op_1, op_2
<instrukcje do asemblacji jeśli <op_1> = <op_2>>
endif
ifdif op_1, op_2
<instrukcje do asemblacji jeśli <op_1> ≠ <op_2>>
endif
ifidni op_1,op_2
<instrukcje do asemblacji jeśli <op_1> = <op_2>>
endif
ifdifi op_1,op_2
<instrukcje do asemblacji jeśli <op_1> ≠ <op_2>>
endif

Różnice pomiędzy powyższymi instrukcjami Ifxxx a IfxxxI jest taka ,że instrukcje IfxxxI ignorują różnice wielkości liter alfabetu przy porównaniu operandów.

8.14 MAKRA

Makro jest jak procedura, która wstawia blok instrukcji w różnych punktach w naszym programie podczas asemblacji. Są trzy główne typy makr w MASM : makra proceduralne ,makra funkcjonalne i makra pętli. Wraz z asemblacją warunkową, narzędzia te dostarczają tradycyjnych konstrukcji if, loop, procedur i funkcji, znanych z wielu języków wysokiego poziomu. W odróżnieniu od instrukcji asemblacji, asemblacja warunkowa i konstrukcje języka makr wykonują się podczas asemblacji, Asemblacja warunkowa i instrukcje makr nie występują kiedy nasz program jest uruchomiony. Celem tych instrukcji jest kontrolowanie ,które instrukcje MASM assembluje do końcowego pliku „.exe" Podczas gdy dyrektywy asemblacji warunkowej wybierają lub pomijają pewne instrukcje do asemblacji, dyrektywy makra pozwalają emitować powtarzające się

sekwencje instrukcji w pliku asemblerowym podobnie jak procedury języków wysokiego poziomu i pętle pozwalające nam powtarzać wykonywanie sekwencji instrukcji języków wysokiego poziomu.

8.14.1 MAKRA PROCEDURALNE

Następująca sekwencja definiuje makro:

```
nazwa       macro       {parametr1 {parametr2 {,...}}}
    <instrukcje>
            endm
```

Nazwa musi być poprawnym i unikalnym symbolem w pliku źródłowym. Będziemy używać tego identyfikatora przy wywoływaniu makra. (Opcjonalne) nazwy parametrów są symbolami zastępczymi dla wartości wyszczególnionych kiedy wywołujemy makro; powyższe nawiasy klamrowe oznaczają pozycje opcjonalne, które nie powinny pojawić się w rzeczywistości w naszym kodzie źródłowym. Te nazwy parametrów są lokalne w makrze i mogą się pojawić gdzie indziej w programie.

Przykład definicji makra:

```
COPY            macro       Przez, Źródło
                mov         ax, Źródło
                mov         Dest, ax
                endm
```

Makro to kopiuje słowo spod adresu źródłowego do słowa pod adresem przeznaczenia. Symbole Przez i Źródło są lokalne w makrze i mogą pojawiać się gdziekolwiek indziej w programie.

Zauważmy, że MASM nie assembluje bezpośrednio instrukcji między dyrektywami macro a endm kiedy MASM napotyka makro. Zamiast tego ,asembler przechowuje teks odpowiadający makru w specjalnej tablicy (nazywanej tablicą symboli).MASM wstawia te instrukcje do naszego programu kiedy wywołujemy makro.

Wywołanie (użycie) makra, polega na wyszczególnieniu nazwy makra jako mnemonika MASMa. Kiedy to zrobimy, MASM wstawi instrukcje pomiędzy dyrektywy macro i endm do naszego kodu a punkcie wywołania makra. jeśli makro ma parametry MASM zastąpi rzeczywiste parametry pojawiające się jako operandy na parametry formalne pojawiające się w definicji makra.. MASM robi bezpośrednie podstawienie tekstowe mimo że stworzyliśmy przyrównanie tekstowe dla parametrów.

Rozważmy następujący kod który używa makra COPY zdefiniowanego powyżej:

```
                call        SetUpX
                copy        Y,X
                add         Y,5
```

Ta część programu będzie wywoływała SetUpX (która ,przypuszczalnie, robi coś ze zmienną X) potem wywołuje makro COPY, które kopiuje wartość w zmiennej X do zmiennej Y.W końcu, dodaje pięć do wartości zawartej w zmiennej Y.

Zauważmy ,że ta sekwencja instrukcji jest absolutnie podobna do:

```
                call        SetUpX
                mov         ax, X
                mov         Y, ax
                add         Y,5
```

W takim przypadku użycie makra może zaoszczędzić znaczną ilość pisaniny w naszym programie. Na przykład przypuśćmy ,że chcemy uzyskać dostęp do elementów różnych dwuwymiarowych tablic. Jak możemy sobie przypomnieć, formuła obliczania adresu rzędowego pozycjonowania elementów dla tablicy to:

Adres elementu = adres bazowy+(Pierwszy Indeks*Rozmiar Rzędu+ Drugi Indeks)* rozmiar elementu

Przypuśćmy, że chcemy napisać kod asemblerowy, który osiągnie ten sam wynik jak następujący kod C:

```
int a[16][7], b[16][7],x[7][16];
int i,j;
    for (i=0; i<16; i=i+1)
        for (j=0; j<7; j=j+1)
            x[j][i] = a[i][j]*b[15-i][j];
```

Kod 8086 dla tej sekwencji jest raczej złożony poprzez liczbę dostępów do tablic. Kompletny kod to:

```
                .386                        ;używamy instrukcji 286 lub 386
                option      segment:use16   ;wymagane dla programów trybu rzeczywistego
                -
                -
                -
a               sword       16 dup (7 dup (?))
b               sword       16 dup (7 dup (?))
x               sword       7 dup (16 dup (?))
```

```
                    -
                    -
                    -
i                   textequ  <cx>               ;przechowanie I w rejestrze cx
j                   textequ  <dx>               ;przechowanie J w rejestrze dx

                    mov     I, 0                ;inicjacja I indeksem pętli zerem
ForILp:             cmp     I, 16               ;czy I jest mniejsze niż 16?
                    jnl     ForIDone            ;jeśli tak skok do treści pętli

                    mov     J, 0                ;inicjacja J indeksem pętli zerem
ForJLp:             cmp     J, 7                ;czy J jest mniejsze iż 7
                    jnl     ForJDone            ;jeśli tak skok do treści pętli J

                    imul    bx,I,7              ;Oblicza indeks dla a[i][j]
                    add     bx, J
                    add     bx, bx              ;Rozmiar elementu jest dwubajtowy
                    mov     ax, A[bx]           ;pobranie a [i][j]

                    mov     bx, 15              ;obliczenie indeksu dla b[15-I][j]
                    sub     bx, I
                    imul    bx, 7
                    add     bx, J
                    add     bx, bx              ;rozmiar elementu jest dwubajtowy
                    imul    ax,b[bx]            ;oblicza a[i][j] * b[16-i][j]

                    imul    bx, J, 16           ;oblicza indeks dla X[j][i]
                    add     bx, I
                    add     bx, bx
                    mov     X[bx], ax           ;przechowuje wynik

                    inc     J                   ;następna iteracja pętli
                    jmp     ForILp
ForJDone:           inc     I                   ;następna iteracji pętli I
                    Jmp     ForILp
ForIDone:
```

Jest to dużo kodu w porównaniu z pięcioma instrukcjami C/C++! Jeśli spojrzymy na tez kod zauważymy, że duża liczba instrukcji po prostu oblicza indeks do trzech tablic. Ponadto sekwencje kodu która oblicza te indeksy tablic są bardzo podobne. Jeśli byłyby takie same, byłoby oczywistym napisanie makra zastępującego obliczanie indeksów trzech tablic. Ponieważ te obliczane indeksy nie są identyczne byłoby świetnie gdyby możliwe było stworzenie makra które uprościłoby ten kod. Odpowiedź brzmi tak :poprzez użycie parametrów makra jest bardzo łatwo napisać takie makro. Rozważmy następujący kod:

```
i           textequ <cx>          ;przechowanie I w rejestrze cx
j           textequ <dx>          ;przechowanie J w rejestrze dx

NDX2        macro   Indeks1,Indeks2,RowSize
            imul    bx, Indeks1, RowSize
            add     bx, Indeks2
            add     bx. Bx
            endm

            mov     I, 0          ;inicjacja pętli I indeksem zero
ForILp:     cmp     I, 16         ;czy I jest mniejsze niż 16
            jnl     ForIDone      ;jeśli tak, skok do treści pętli I

            mov     J, 0          ;inicjacja pętli J indeksem zero
ForJLp:     cmp     J, 7          ;czy J jest mniejsze niż 7?
            jnl     ForJDone      ;jeśli tak, skok do treści pętli J
```

```
              NDX2  I,J,7
              mov   ax, A[bx]        ;pobranie a[i][j]
              mov   bx, 15           ;obliczanie indeksu dla b[15-I][j]
              sub   bx, I
              NDX2  bx,J,7
              imul  ax,b[bx]         ;Oblicza a[i][j]*b[15-i][j]

              NDX2  J,I,16
              mov   X[bx], ax        ;przechowanie wyniku

              inc   J                ;następna iteracja pętli J
              jmp   ForJLp
ForJDone:     inc   I                ;następna iteracja pętli I
              Jmp   ForILp
ForIDone:
```

Jeden problem z makrem NDX2 jest taki, że musimy znać rozmiar wiersza tablicy (ponieważ jest to parametr makra).W krótkim przykładzie, jak ten, nie jest to duży problem. Jednak, jeśli piszemy duży program możemy łatwo zapomnieć rozmiary i musimy szukać ich, lub co gorsza, "zapamiętać" je niepoprawnie i wprowadzić błąd do naszego programu. jednym sensownym pytaniem jest czy MASM może wykombinować rozmiar wiersza tablicy automatycznie. Odpowiedź brzmi tak.

Operator MASMa length jest przypuszczalnie zwraca liczbę elementów w tablicy. Jednakże wszystkie rzeczywiste zwroty odnoszą się do pierwszej wartości pojawiającej się w polu operandu tablicy. Na przykład, (length a) zwróci 16 z danej powyższej definicji. MASM poprawi ten problem przez wprowadzenie operatora lengthof, który. Właściwie zwróci całkowitą liczbę elementów w tablicy.(Lengthof a),na przykład, właściwie zwróci 112 (16*7).Chociaż operator (length a) zwraca błędną wartość dla naszego celu (zwraca rozmiar kolumny zamiast wiersza),możemy użyć tej zwracanej wartości do obliczenia rozmiaru wiersza używając wyrażenia (lengthof a)/(length a).Z taką wiedzą rozważmy następujące dwa makra:

```
;LDAX- Jest to makro ładujące ax słowem spod adresu Array[Index1][Index2]
;       Założenie:     Zadeklarujemy tablicę używając instrukcji takiej jak
;                             Array word Colsize dup (Rowsize dup (?))
;                      a tablica jest przechowana w rzędowym przechowywaniu elementów.
;       Jeśli wyszczególnimy (opcjonalnie) czwarty parametr, jest to instrukcja maszynowa 8086 zastępowana
;       przez instrukcję MOV ,która ładuje AX z Array[bx]
LDAX                   macro  Array, Index1,Index2,Instr
                       imul   bx,Index1, (lengthof Array) / (length Array)
                       add    bx, Index2
                       add    bx, bx
;zobaczmy czy jest dostarczony czwarty parametr
                       ifb    <Instr>
                       mov    ax, Array[bx]          ;Jeśli nie, emituj instrukcję mov
                       else
                       instr  ax, Array[bx]          ;jeśli tak, emituj instrukcję użytkownika
                       endif
                       endm
;STAX –jest to makro przechowujące ax w słowie spod adresu Array[Index1][Index2]
;       Założenie:     Takie jak powyżej
STAX                   macro  Array,Index1,Index2
                       imul   bx, Index1, (lengthof Array)/(length Array)
                       add    bx, Index2
                       add    bx, bx
                       mov    Array[bx], ax
                       endm
```

Z powyższymi makrami, oryginalny program będzie się przedstawiał tak:

```
i             textequ <cx>           ;przetrzymanie I w rejestrze cx
j             textequ <dx>           ;przetrzymanie J w rejestrze dx

              mov   I,0              ;inicjacja pętli I  indeksem zero
ForILp:       cmp   I, 16            ;czy I jest mniejsze niż 16?
              jnl   ForIDone         ;jeśli tak skocz do treści pętli I
```

```
            mov     J ,0                    ;inicjacja pętli J indeksem zero
ForJLp:     cmp     J, 7                    ;czy J jest mniejsze niż 7?
            jnl     ForJDone                ;jeśli tak, skocz do treści pętli J

            ldax    A,I, J                  ;pobranie A[I][J]
            mov     bx,16
            sub     bx,I                    ;obliczanie 16-I
            ldax    b, bx,J, imul           ;mnożenie B[16-I][J]
            stax    x, J, I                 ;przechowanie X[J][I]

            inc     J                       ;następna iteracja pętli
            jmp     ForJLp
ForJDone:   inc     I                       ;następna iteracja pętli I
            Jmp     ForILp
ForIDone:
```

Jak widać wyraźnie, kod z powyższymi pętlami jest krótszy poprzez użycie tych makr. Oczywiście, cała sekwencja kodu jest w rzeczywistości dłuższa ponieważ makra przedstawiają więcej linii kodu, który zachowują w oryginalnym programie. jednakże, jest to artefakt tego szczególnego programu. Generalnie ,prawdopodobnie będziemy mieli więcej niż trzy tablice; co więcej, możemy zawsze dołożyć LDAX i STAX do pliku bibliotecznego i automatycznie wprowadzać je zawsze tam gdzie zajmujemy się dwuwymiarowymi tablicami. Chociaż, technicznie ,nasz program może w rzeczywistości zawierać więcej instrukcji assemblera, jeśli wprowadzamy te makra w naszym kodzie, musimy tylko napisać te makra jeden raz. Zresztą zajmuje niewiele wysiłku wprowadzenie makr do każdego nowego programu.

Możemy skrócić tę sekwencję kodu nawet bardziej używając kilku dodatkowych makr. Jednakże, jest kilka dodatkowych tematów do omówienia zanim będziemy mogli to zrobić.

8.14.2 MAKRA A PROCEDURY 80x86

Początkujący programiści asemblerowi często mylą makra i procedury. Procedura jest pojedynczą częścią kodu, którą wywołujemy z różnych punktów programu. Makro jest sekwencją instrukcji, które MASM kopiuje w naszym programie ilekroć używamy makra. Rozważmy dwa fragmenty kodu:

```
Proc_1      proc        near
            mov         ax, 0
            mov         bx, ax
            mov         cx, 5
            ret
Proc_1      endp

Macro_1     macro
            mov         ax, 0
            mov         bx, ax
            mov         cx, 5
            endm

            call        Proc_1
            -
            -
            call        Proc_1
            -
            -
            Macro_1
            -
            -
            Macro_1
```

Chociaż makro i procedura dają ten sam wynik, robią to na różne sposoby. Procedura generuje kod kiedy assembler napotka dyrektywę proc. Wywołanie tej procedury wymaga tylko trzech bajtów. W czasie wykonania,80x86:

- napotyka instrukcję call
- odkłada adres powrotu na stos

- skacze do Proc_1
- w tym wykonuje kod
- zdejmuje ze stosu adres powrotu
- wraca do kodu wywołującego

Makro, z drugiej strony, nie emituje żadnego kodu kiedy przetwarza instrukcje pomiędzy dyrektywami macro i endm. Jednak, po napotkaniu Macro_1 w polu mnemoniku ,MASM będzie assemblował każdą instrukcję pomiędzy dyrektywami macro i endm i emitował ten kod do pliku wyjściowego. W czasie wykonywania, CPU wykonuje te instrukcje bez obciążenia call /ret.

Wykonanie makrorozwinięcia jest zazwyczaj szybsze niż wykonanie tego samego kodu implementowanego w procedurze. Jednakże, jest to inny przykład klasycznego problemu szybkość/ miejsce. Makra wykonują się szybciej poprzez wyeliminowanie sekwencji call /return. jednak asembler kopiuje kod makro do naszego programu przy każdym wywołaniu makra. Jeśli mamy dużo wywołań makra wewnątrz programu, będzie on dużo większy niż ten sam program ,który używa procedur.

Wywołania makr i wywołania procedur znacznie się różnią. Aby wywołać makro, musimy po prostu wyszczególnić nazwę makra jak gdyby była instrukcją lub dyrektywą. Do wywołania procedury musimy użyć instrukcji call. W wielu wypadkach jest nieszczęśliwie ,że używamy dwóch oddzielnych mechanizmów wywołań dla tak podobnych operacji. Rzeczywisty problem występuje jeśli chcemy przełączyć makro na procedurę lub vice versa. Może być tak, że używaliśmy makrorozwinięcia dla szczególnej operacji, ale teraz rozwinęliśmy makro tak wiele razy, że większego sensu nabiera zastosowanie procedury. Być może właśnie przeciwieństwo jest prawdą, użyliśmy procedury, ale chcemy rozszerzyć kod wplatany do poprawienia jej wydajności. Problem z jedną i drugą konwersja jest taki, że będziemy musieli znaleźć każde wywołanie makro lub procedury i zmodyfikować je. Modyfikacja makra lub procedury jest łatwa, ale zlokalizowanie i zmiana wszystkich wywołań może dać trochę pracy. Na szczęście, jest bardzo prosta technika, którą możemy użyć jako wywołania procedury dzieląc składnię z wywołaniem makra. Sztuczka ta to stworzenie makra lub równania tekstowego dla każdej procedury jaką piszemy, która rozszerza ją o wywołanie tej procedury. Na przykład przypuśćmy, że napiszemy procedurę ClearArray, która zeruje tablice. Kiedy napiszemy kod, możemy zrobić to następująco.:

```
ClearArray          textequ  <call $$ClearArray>
$$ClearArray        proc     near
                    -
                    -
$$ClearArray        endm
```

Wywołanie procedury ClearArray używa po prostu instrukcji takiej jak ta:

```
                    -
                    -
                    -
        <Ustaw parametry dla ClearArray>
                    ClearArray
                    -
                    -
                    -
```

Jeśli kiedyś zmienimy procedurę $$ClearArray na makro, wszystko co musimy zrobić to nazwać ClearArray i pozbyć się textequ dla procedury. Odwrotnie, jeśli już mamy makro i chcemy skonwertować je na procedurę, po prostu nazywamy procedurę $$procname i tworzymy przyrównanie tekstowe, które emituje wywołanie do tej procedury. Pozwala to nam na zastosowanie tej samej składni wywołania dla procedur i makr.

Ten tekst nie używa normalnie omówionych powyżej technik, z wyjątkiem podprogramów Standardowej Biblioteki UCR. Jest tak ponieważ nie jest to dobry sposób na wywoływanie procedur. Niektórzy ludzie mają problemy w odróżnieniu makr i procedur, więc ten tekst będzie używał wyraźnych wywołań pomagając unikać takich pomyłek .Wywołania Biblioteki Standardowej są wyjątkiem ponieważ stosowanie wywołań makr jest standardowym sposobem wywołania tych podprogramów.

8.14.3 DYREKTYWA LOCAL

Rozważmy następującą definicję makra:

```
LJE         macro   Przez
            jne     SkipIt
            jmp     Przez
SkipIt:
            Endm
```

To makro robi „długi skok jeśli równe". Jednakże, jest jeden problem z nim. Ponieważ MASM kopiuje teks makra dosłownie, symbol SkipIt będzie przedefiniowany za każdym razem, kiedy makro LJE się pojawia. Kiedy to się wydarzy, asembler wygeneruje wielokrotnie błąd definicji. Przezwyciężeniem tego problemu jest dyrektywa local, która może być zastosowana do zdefiniowania lokalnego symbolu wewnątrz makra. Rozważmy następującą definicję makra:

```
LJE         macro   Przez
            local   SkipIt
            jne     SkipIt
            jmp     Przez
SkipIt:
            Endm
```

W tej definicji makra, SkipIt jest symbolem lokalnym. Dlatego też, asembler wygeneruje nową kopię SkipIt za każdy razem ,kiedy jest wywoływane makro. Zapobiega to generowaniu przez MASM błędów.

Dyrektywa local, jeśli pojawi się wewnątrz naszej definicji makra, musi pojawić się bezpośrednio po dyrektywie macro .Jeśli potrzebujemy wielorakich lokalnych symboli ,możemy wyszczególnić kilka z nich w polu operandu dyrektywy local, po prostu oddzielając każdy symbol przecinkiem:

```
IFEQUAL          macro   a , b
                 local   ElsePosition, Done
                 mov     ax , a
                 jne     ElsePosition
                 inc     bx
                 jmp     Done
ElsePosition:    dec     bx
Done:
                 endm
```

8.14.4 DYREKTYWA EXITM

Dyrektywa exitm bezpośrednio zakańcza makro, dokładnie jak gdyby MASM napotkał endm. MASM ignoruje cały tekst od dyrektywy exitm do endm.

Prawdopodobnie zastanawiacie się dlaczego ktoś miałby używać dyrektywy exitm .W końcu jeśli MASM ignoruje cały tekst między exitm i endm, dlaczego zawracać sobie głowę umieszczaniem dyrektywy exitm w naszym makrze na pierwszym miejscu? Odpowiedzią jest asemblacja warunkowa. Asemblacja warunkowa może być zastosowana do warunkowego wykonania dyrektywy exitm, tym samym pozwalając na dalsze makrorozwinięcia pod pewnymi warunkami, rozważmy coś takiego:

```
Bytes       macro   Count
            byte    Count
            if      Count  eq  0
            exitm
            endif
            byte    Count dup (?)
            endm
```

Oczywiście ten prosty przykład może być zakodowany bez zastosowania dyrektywy exitm, ale on demonstruje jak dyrektywa exitm może być użyta wewnątrz sekwencji asemblacji warunkowej do sterowania jego oddziaływaniem.

8.14.5 PARAMETRY MAKROROZWINIĘCIA I OPERATORY MAKRA

Ponieważ MASM wykonuje tekstowe zastąpienie dla parametrów makra, kiedy wywołujemy makro, są chwile, kiedy wywołanie makra może nie przynieść wyniku jakiego oczekujemy. Na przykład rozpatrzmy następującą (co prawda głupią) definicję makra:

```
Index       =           8
;Problem -  Makro to próbuje załadować AX elementem tablicy słów, wyszczególnionej przez parametr
;           makra. Parametr ten musi być stałą w czasie asemblacji

Problem     macro       Parametr
            mov         ax, Array[Parametr*2]
            endm
```

```
-
-
-
Problem        2
-
-
-
Problem        Index+2
```

Kiedy MASM rozwija pierwsze wywołanie Problem ,tworzy instrukcje:

```
mov        ax, Array[2*2]
```

Okay ,jak dotąd wszystko dobrze. Kod ten ładuje element dwa tablicy Array do ax. Jednak, rozważmy rozwinięcie drugiego wywołania Problem:

```
mov        ax, Array[Index+2*2]
```

ponieważ wyrażenia adresowe MASMa wspierają operatory pierwszeństwa ,to makrorozwinięcie nie da prawidłowego wyniku. Uzyskamy dostęp do szóstego elementy Array (pod indeksem 12) zamiast do elementu dziesiątego spod indeksu 20.

Powyższy problem wystąpi ponieważ MASM po prostu zastąpi parametr formalny przez tekst rzeczywistego parametru a nie wartość rzeczywistego parametru. Taki mechanizm przekazywanie przez nazwę powinien być znany od dawna programistom C i C++, którzy używają instrukcji #define. Jeśli sądzimy, że parametry makra (przekazywanie przez nazwę) pracuje tak jak Pascalowskie lub C przekazywanie parametrów przez wartość, musimy nastawić się na końcowe nieszczęście.

Jednym możliwym rozwiązaniem, które działa dla makr takich jak powyższe, jest wstawienie nawiasów okrągłych wokół parametrów makra, które występują wewnątrz wyrażenia w środku makra. Rozpatrzmy następujący kod:

```
Problem    macro      Parametr
           mov        ax, Array[{Parametr)*2]
           endm
           -
           -
           -
           Problem    Index+2
```

Makro to wywołuje rozwinięcie

```
           Mov        ax, Array[(Index+2)*2]
```

To daje nam spodziewany rezultat.

Parametr tekstowy jest zastępowany ale jest jeden problem w działaniu kiedy używamy makr. Innym problemem występuje ponieważ MASM ma dwa typy wartości czasu asemblacji: numeryczne i tekstowe. Niestety MASM oczekuje wartości numerycznych w pewnym kontekście i wartości tekstowych w innym. Nie są one w pełni zamienne. Na szczęście MASM dostarcza zbioru operatorów, które pozwalają nam konwertować pomiędzy jedną a drugą formą (jeśli jest to możliwe do wykonania).Aby zrozumieć subtelne różnice pomiędzy tymi dwoma typami wartości spójrzmy na poniższe instrukcje:

```
Numeryczne    =          10+2
Tekstowe      textequ    <10+2>
```

MASM ocenia wyrażenie numeryczne „10+2” i łączy wartość dwanaście z symbolem Numeryczny. Dla symbolu Tekstowy po prostu przechowuje łańcuch „10+2” i zastępuje go dla Tekstowy gdziekolwiek użyjemy go w wyrażeniu.

W wielu kontekstach możemy użyć jednego z dwóch symboli. Na przykład, poniższe dwie instrukcje obie ładują dwanaście do ax:

```
mov    ax, Numeryczny      ;to samo co mov ax, 12
mov    ax, Tekstowy        ;to samo co mov ax,10+2
```

Jednak rozważmy poniższe dwie instrukcje:

```
mov    ax, Numeryczny*2    ;to samo co mov ax, 12*2
mov    ax, Tekstowy*2      ;to samo co mov ax, 10+2*2
```

Jak widzimy, zastąpienie tekstu, które występuje z przyrównaniem tekstowym może prowadzić do tego samego problemu, który napotykamy przy zastępowaniu tekstowym parametrów makr.

MASM automatycznie konwertuje obiekt tekstowy na wartość numeryczną, jeśli konwersja jest konieczna. Jeśli omówiliśmy problem zastępowania tekstowego, możemy użyć wartości tekstowej (jeśli ciąg przedstawia wielkość liczbową) gdziekolwiek MASM wymaga wartości liczbowej.

Idąc w przeciwnym kierunku, konwersja wartości liczbowej na tekstową nie jest automatyczna. Dlatego też MASM dostarcza operatora, którego możemy użyć do konwersji danej liczbowej do danej tekstowej:

operator „%”.Ten operator rozszerzenia wymusza bezpośrednie wyliczenie wyrażenia potem konwertuje wynik wyrażenia do ciągu cyfr. Spójrzmy na wywołanie makra Problem:

```
                Problem     10+2        ;Parametrem jest „10+2”
                Problem     %10+2       ;Parametrem jest „12”
```

W drugim powyższym przykładzie operator rozwinięcia tekstu instruuje MASM aby wyliczył wyrażenie „10+2” i konwertuje wynik wartości liczbowej do wartości tekstowej składającej się z cyfr, które przedstawiają wartość dwanaście. Dlatego też, te dwa makra rozwijają się (odpowiednio) w następujące instrukcje:

```
                mov     ax, Array[10+2*2]
                mov     ax, Array[12+2]
```

MASM dostarcza drugiego operatora, operatora zastąpienia, który pozwala nam rozwinąć nazwy parametrów makra gdzie MASM normalnie nie oczekuje symbolu. Operator zastąpienia to znak „&” (ampersand) .Jeśli otoczymy nazwę parametru makra ampersandami wewnątrz makra ,MASM zastąpi parametr tekstowy bez względu na lokację symbolu. Pozwala to nam rozwinąć parametry nazw których nazwy pojawiają się wewnątrz innych identyfikatorów lub wewnątrz ciągów literowych. Poniższe makro demonstruje zastosowanie tego operatora:

```
DebugMsg            macro           Point,String
Msg&String&         byte            „At point &Point&: &String&”
                    endm
                    -
                    -
                    -
                    DebugMsg  5,  <Twierdzenie błędne>
```

Makro wywołuje bezpośrednio instrukcję:

```
Msg5                byte            „At point  5:  Twierdzenie błędne”
```

Zauważmy, że operator zastąpienia pozwolił temu makru na połączenie :Msg” i „5” w etykietę dyrektywy bajtowej. Zauważmy również ,że operator rozwinięcia pozwala nam rozwinąć identyfikator makra nawet jeśli pojawia się on w stałym ciągu literowym. Bez ampersandu w ciągu ,MASM wyemitowałby instrukcję:

```
Msg5                byte            „At point : String”
```

Innym ważnym operatorem aktywnym wewnątrz makr jest operator dosłownego znaku, wykrzyknik „!”.Symbol ten instruuje MASM, żeby przekazał znak bez żadnych modyfikacji. Normalnie będziemy używać tego symbolu jeśli musimy objąć jeden z następujących symboli jako znaków wewnątrz makra:

```
                    !       &       >       %
```

Na przykład, mamy ochotę wyświetlić w ciągu makra DebugMsg ampersand, użyjemy definicji:

```
DebugMsg            macro           Point,String
Msg&String&         byte            „At point !&Point!&: &String&”
                    Endm
```

„debug 5, <Twierdzenie błędne> dostarczy następującej instrukcji:

```
Msg5                byte            „At point &Point&:  &String&”
```

Użycie symboli „<” i „>” ogranicza dane tekstowe wewnątrz MASMa. Poniższe dwa wywołania makra PutData pokazują jak możemy zastosować te ograniczniki w makrze:

```
PutData         macro           Nazwa, Dane
PD_&Nazwa&      byte            Dane
                endm
                -
                -
                -
                PutData         MDane, 5,4,3            ,emituje „PD_Mdane byte 5”
                PutData         MDane  <5,4,3>          ;emituje „PD_Mdane byte  5,4,3
```

Możemy użyć ograniczników tekstu do otoczenia obiektów, które życzymy sobie traktować jako pojedynczy parametr zamiast jako listę wielu parametrów. W przykładzie PutData, pierwsze wywołanie przekazuje cztery parametry do PutData (PutData ignoruje ostatnie dwa)W drugim wywołaniu, są dwa parametry, drugi składa się z tekstu 5,4,3.

Ostatnim interesującym nas operatorem makra jest operator „;:”.operator ten zaczyna komentarz makra. Zwykle MASM kopiuje cały tekst z makra do treści programu podczas asemblacji, wliczając w to wszystkie komentarze. Jednak, jeśli zaczniemy komentarz od „ ;;” zamiast pojedynczego średnika, MASM nie rozwinie tego komentarza jako części kodu podczas makrorozwinięcia. Zwiększa to szybkość asemblacji a co ważniejsze, nie zaśmieca listingu programu kopiami tych samych komentarzy (zobacz „Kontrola Listingu” aby nauczyć się więcej o listingach programu)

Operator	Description
&	Text substitution operator
< >	Literal text operator
!	Literal character operator
%	Expression operator
;;	Macro comment

Tablica 42: Operatory Makra

8.14.6 PRÓBKA MAKRA IMPLEMENTUJĄCEGO PĘTLĘ FOR

Pamiętamy operacje dla pętli i macierzy używane w poprzednim przykładzie? W konkluzji tej sekcji był krótki komentarz, że możemy „poprawić" ten kod bardziej, używając makr, ale na przykład musimy poczekać. Korzystając z opisu operatorów makra, możemy skończyć teraz to omówienie. Makra, które są zaimplementowane dla pętli for:

```
;Po pierwsze trzy makra, które pozwolą nam skonstruować symbole poprzez łączenie innych. Jest to konieczne
;ponieważ kod ten musi rozszerzyć kilka komponentów w tekście przyrównywanym wiele razy aby dojść do
;poprawnego symbolu
;
;MakeLbl          - Emituje tworzenie etykiety prze połączenie dwóch parametrów przekazanych do tego makra.
MakeLbl              macro          FirstHalf,SecondHalf
&FirstHalf&&SecondHalf&:
                     endm
jgDone               macro          FirstHalf,ScondHAlf
                     jg             &FirstHAlf&&SecondHAlf&
                     endm
;ForLp            - Makro to pojawia się na początku  pętli for. Wywołując to makro, używamy instrukcji:
;
;                    ForLp          LoopCtrlVar, StartVal, StopVal
;
;Notka "FOR" jest w MASM słowem zarezerwowanym, dlatego to makro nie używa tej nazwy.
ForLp                macro          LCV,Start,Stop
;Musimy wygenerować unikalny globalny symbol dla każdej pętli for jaką tworzymy. Symbol te musi być
;globalny, ponieważ będziemy musieli odnosić się do niego u podstawy pętli. Generując unikalny symbol makro
;to łączy „FOR" z  nazwą zmiennej sterującej pętli i unikalną wartością liczbową, którą to makro zwiększa za
;każdym razem, kiedy użytkownik konstruuje pętlę for z takiej samej zmiennej sterującej pętli.
                     ifndef         $$For&LCV&     ;;Symbol =$$FOR połączony z LCV
$$For&LCV&           =              0              ;;Jeśli jest to pierwsza pętla / LCV używa
                     else                          ;;zera, inaczej zwiększa wartość
$$For&LCV&           =              $$For&LCV&+1
                     endif
;Emisja instrukcji do inicjacji zmiennej sterującej pętli:
                     mov            ax, Start
                     mov            LCV, ax
;etykieta wyjściowa na szczycie pętli .Przybiera postać
                     $$FOR  LCV  x
;gdzie LCV jest nazwą zmiennej sterującej pętli a X jest unikalnym numerem, które to makro zwiększa dla
;każdej pętli for, która używa takiej samej zmiennej sterującej pętli.
                     MakeLbl        $$For&LCV&, %$$For&LCV&
;Okay, kod wyjściowy dla tej pętli for jest kompletny.
;Makro jgDone generuje skok (jeśli większy) do etykiety Next makro emituje poniższy kod
                     mov            ax, LCV
                     cmp            ax, Stop
                     jgDone         $$Next&LCV&, %$$For&LCV&
```

;Makro Next kończy pętlę for. Makro to zwiększa zmienną sterującą pętli a potem przekazuje sterowanie powrotnie do etykiety na szczycie pętli for

```
Next                macro       LCV
                    inc         LCV
                    jmpLoop     $$For&LCV&, %$$For&LCV&
                    MakeLbl     $$Next&LCV&, %$$For$LCV&
                    endm
```

Z tymi makrami i makrami LCAX/STAX,kod prezentowany wcześniej do manipulowania tablicą ,staje się bardzo prosty

```
                    ForLp       I,0,15
                    ForLp       J,0,6

                    ldax        A,I,J           ;pobiera A[I]{J}
                    mov         bx, 15          ;oblicza 16 - I
                    sub         bx, I
                    ldax        b, bx,J, imul   ;mnoży w B[15-i][J}
                    stax        x, J,I          ;przechowuje X [J][I]

                    Next        J
                    Next        I
```

Chociaż kod ten nie jest całkiem tak krótki jak oryginalny przykład w C/C++, jest już całkiem znośny.

Podczas gdy program główny stał się dużo prostszy, jest pytanie o same makra. Makra ForLp i Next są niezmiernie złożone! Gdybyśmy musieli wkładać taki wysiłek za każdym razem, kiedy chcemy stworzyć makro, program asemblerowy byłby dziesięć razy cięższy do napisania, jeśli zdecydowalibyśmy się na użycie makr. Na szczęście ,musimy tylko raz napisać (i zdebuggować) makro takie jak to. Wtedy możemy go używać tak wiele razy jak chcemy, w wielu różnych programach bez każdorazowego martwienia się o jego implementację.

Mając złożoność makr For i Next, prawdopodobnie jest dobrym pomysłem ostrożnie opisać co każda instrukcja w tych makrach robi. Jednak, przed omówieniem makr powinniśmy omówić dokładnie jak można zaimplementować pętle for / next w języku asemblera. Ten tekst w pełni bada pętlę for trochę później ,ale możemy z pewnością przejść tutaj podstawy. Rozważmy następującą Pascalowska pętle for:

```
          for zmienna := Start to End    do
                    Jakieś_Instrukcje;
```

Pascal zaczyna od obliczenia wartości Start. Wtedy przydziela tą wartość do zmiennej sterującej pętli (zmienna).Potem oblicza End i zachowuje tą wartość w lokacji tymczasowej. Wtedy instrukcja pascalowska for jest wprowadzana do treści pętli. Pierwszą rzeczą jaką robi pętla jest porównanie wartości zmienna z wartością obliczoną dla End.Jeśli wartość zmienna jest większa niż wartość End,Pacal przechodzi do pierwszej instrukcji po pętli for, w przeciwnym razie wykonuje Jakieś_Instrukcje. Po wykonaniu przez pętlę for Jakieś_Instrukcje, dodawana jest jedynka do zmienna i wykonywany skok do punktu gdzie porównywana jest wartość zmienna z obliczoną wartością End. Konwertując ten kod bezpośrednio na język asemblera mamy następujący kod:

```
;Notka: Ten kod zakłada że Start i End są prostymi zmiennymi. Jeśli tak jest, obliczamy wartość dla tych wyrażeń
;i umieszczamy je w tych zmiennych
            mov         ax, Start
            mov         Zmienna, ax
ForLoop:    mov         ax, Zmienna
            cmp         ax, End
            jg          ForDone
        <Kod dla Jakieś_Instrukcje>

            inc         Zmienna
            jmp         ForLoop
ForDone:
```

Aby implementować to jako zbiór makr, będziemy musieli napisać krótki kawałek kodu który napisze powyższe instrukcje asemblera dla nas. Na pierwszy rzut oka, wydawałoby się to łatwe, dlaczego nie użyć następującego kodu?

```
ForLp       macro       Zmienna, Start, Stop
            mov         ax, Start
```

```
            mov         Zmienna, ax
ForLoop:    mov         ax, Zmienna
            cmp         ax, Stop
            jg          ForDone
            endm
Next        macro       Zmienna
            inc         Zmienna
            jmp         ForLoop
ForDone:
            Endm
```

Makra te stworzyłyby poprawny kod – dokładnie raz. jednak problem rozwinąłby się gdybyśmy spróbowali użyć tych makr po raz drugi .jest to szczególnie widoczne kiedy używamy pętli zagnieżdżonych:

```
            ForLp       I, 1, 10
            ForLp       J, 1, 10
            -
            -
            -
            Next        J
            Next        I
```

Powyższe makra emitują kod 80x86:

```
            mov         ax, 1               ;makro ForLp I, 1 ,10 emituje te
            mov         I, ax               ;instrukcje
ForLoop     mov         ax, I               ;       -
            cmp         ax, 10;             ;       -
            jg          ForDone             ;       -
                                            ;
            mov         ax, 1               ;Makro ForLp J, 1 ,10 emituje te
            mov         J, ax               ;instrukcje
ForLoop     mov         ax, J               ;       -
            cmp         ax, 10              ;       -
            jg          ForDone             ;       -
            -
            -
            -
            inc         J                   ;makro Next emituje te instrukcje
            jmp         ForLp               ;
ForDone:
            inc         I                   ;makro Next J emituje te instrukcje
            jmp         ForLp               ;
ForDone:
```

Problem, widoczny w powyższym kodzie, jest taki za każdym razem kiedy używamy makra ForLp, emitujemy etykietę „ForLoop" do kodu. Podobnie, za każdym razem, kiedy używamy makra Next, emitujemy etykietę „ForDone" do strumienia kodu. Dlatego też, jeśli używamy tych makr więcej niż raz (wewnątrz tej samej procedury) dostaniemy powielony symbol błędu. Aby zapobiec temu błędowi, makra musza wygenerować unikalne etykiety za każdym razem kiedy ich używamy. Niestety, dyrektywa local nie robi tego. Dyrektywa local definiuje unikalny symbol wewnątrz pojedynczego wywołania makra. Jeśli spojrzymy uważnie na powyższy kod, zobaczymy, że makro ForLp emituje symbol do którego odnosi się kod w makrze Next. Podobnie, makro Next emituje etykietkę do której odnosi się makro ForLp .Dlatego też, nazwa etykiety musi być globalna ponieważ dwa makra mogą odnosić się wzajemnie do etykiet.

Rzeczywistym rozwiązaniem zastosowania makr ForLp i Next jest wygenerowanie znanych globalnie etykiet w postaci „$$For"+"nazwa zmiennej" + „jakiś unikalny numer" i „$$Next+"nazwa zmiennej"+"jakiś unikalny numer". Dla przykładu podanego wyżej, rzeczywiste makra ForLp i Next generują następujący kod:

```
            mov         ax, 1       ;Makro ForLp I, 1, 10 emituje te instrukcje
            mov         I,ax        ;
$$ForIO:    mov         ax, I       ;
            cmp         ax, 10      ;
            jg          $$NextIO    ;
            mov         ax, 1       ;Makro ForLp J, 1 ,10 emituje te instrukcje
            mov         J, ax       ;
```

```
            mov         ax, J           ;
            cmp         ax, 10          ;
            jg          $$NextJO        ;
            -
            -
            -
            inc         J               ;Makro Next emituje te instrukcje
            jmp         $$ForJO         ;
$$NextJO:
            inc         I               ;Makro I emituje te instrukcje
            jmp         $$ForIO
$$NextIO:
```

Pozostaje pytanie „Jak wygenerować takie etykiety?”

Budowa symbolu w postaci „$$ForI” lub „$$NextJ” jest stosunkowo łatwa. Tworzymy ten symbol poprzez połączenie ciągu „$$For” lub „$$Next” z nazwą zmiennej sterującej pętli .Problem wystąpi wtedy, kiedy spróbujemy dołączyć wartość liczbową na koniec tego ciągu. rzeczywisty kod ForLp i Next osiągnie to stworzenie nazwy zmiennej w postaci „$$For nazwa zmiennej” w czasie asemblacji i zwiększy tą zmienną dla każdej pętli z daną nazwą zmiennej sterującej. Poprzez wywołanie makr MakeLbl,jgDone i jmpLoop,ForLp i Next otrzymamy właściwe etykiety i pomocnicze instrukcje.

Makra ForLp i Next są dosyć złożone. Daleko bardziej złożone, niż te które znajdziemy w programach. Demonstrują one jednak siłę i możliwości makr MASMa. Nawiasem mówiąc, dużo lepszym sposobem do tworzenia tych symboli jest zastosowanie funkcji makra.

8.14.7 FUNKCJE MAKRA

Funkcja makra jest makrem ,którego jedynym celem jest zwracanie wartości do użycia w polu operandu jakiejś innej instrukcji. Chociaż jest oczywiste podobieństwo pomiędzy procedurami a funkcjami w językach wysokiego poziomu a makrami proceduralnymi i funkcjonalnymi, analogia jest daleko bardziej zupełna. Funkcje makra nie pozwalają nam na stworzenie sekwencji kodu ,który emituje jakieś instrukcje, które obliczają wartości, kiedy program się wykonuje. Zamiast tego, funkcje makra po prostu obliczają jakąś wartość w czasie asemblacji, kiedy MASM może używać operandu.

Dobrym przykładem funkcji makra jest funkcja Date.Makro to pakuje wartość pięciu bitów dnia, czterech bitów miesiąca i siedem bitów roku do wartości 16 bitów i zwraca tą wartość 16 bitową jako wynik .Jeśli musielibyśmy stworzyć zainicjowaną tablicę dat, moglibyśmy użyć poniższego kodu:

```
DateArray           word    Date (2, 4, 84)
                    word    Date (1, 1, 94)
                    word    Date (7, 20, 60)
                    word    Date (7, 19, 69)
                    word    Date (6, 18, 74)
                    -
                    -
                    -
```

Funkcja Date spakuje dane a dyrektywa word wyemituje 16 bitową spakowaną wartość dla każdej daty pliku wynikowego. Wywołujemy funkcje makra poprzez zastosowanie ich nazw ,gdzie MASM oczekuje jakiegoś rodzaju wyrażenia tekstowego. Jeśli funkcja makra wymaga parametrów musimy otoczyć je wewnątrz nawiasami ,podobnie jak parametry Date.

Funkcje makra wyglądają dokładnie jak makra standardowe z dwoma wyjątkami: nie zawierają instrukcji, które generują kod i zwracają wartość tekstową poprzez operand do dyrektywy exitm .Zauważmy, że nie możemy zwrócić wartości liczbowej z funkcji makra. Jeśli musimy zwrócić wartość liczbową ,najpierw musimy skonwertować ją do wartości tekstowej.

Następująca funkcja makra implementuje Date używając 16 bitowego formatu danych podanych w Rozdziale Pierwszym.

```
Date        macro       month, day, year
            local       Value
Value       =           (month shl 12) or (day shl 7) or year
            Exitm       %Value
            Endm
```

Operator wyrażenia („%”) jest konieczny w polu operandu dyrektywy exitm ponieważ funkcje makra zawsze zwracają dane tekstowe a nie numeryczne. Operator wyrażenia konwertuje wartość liczbową do ciągu cyfr do przyjęcia przez exitm.

Jeden drobny problem z powyższym kodem jest taki, że funkcja zwraca śmieci jeśli data jest nieprawidłowa. Lepszym rozwiązaniem będzie wygenerowanie błędu jeśli dane wejściowe będą niepoprawne. Możemy użyć dyrektywy „.err” i zrobić asemblację warunkową. Poniższa implementacja Date sprawdza wartości miesiąca, dnia i roku aby zobaczyć czy są one sensowne:

```
Date        macro       monyh,day, year
            local       Value

            if          (month gt 12)  or (month lt 1) or \
                        (day gt 31) or (day lt 1) or \
                        (year  gt 99) or (year lt 1)
                        .err
                        exitm  <0>               ;;musi zwrócić coś!
                        endif
Value                   =      (month shl 12) or (day shl 7) or year
                        exitm  %Value
                        endm
```

W tej wersji, każda próba wprowadzenia nieprawidłowej daty wywoła dyrektywę .err, która wywoła błąd w czasie asemblacji.

8.14.8 MAKRA PRZEDEFINIOWANE,FUNKCJE MAKRA I SYMBOLE

MASM dostarcza czterech wbudowanych makr i czterech odpowiednich funkcji makra. W dodatku, MASM również dostarcza dużej liczby uprzednio zdefiniowanych symboli, do których możemy uzyskać dostęp podczas asemblacji. Chociaż będziemy rzadko używali tych makr ,funkcji i zmiennych dla w miarę złożonych makr, są one niezbędne, kiedy ich potrzebujemy

Name	operands	Example	Description
substr	string, start, length Returns: text data	NewStr substr Oldstr, 1, 3	Returns a string consisting of the characters from start to start+length in the string operand. The length operand is optional. If it is not present, MASM returns all characters from position start through the end of the string.
instr	start, string, substr Returns: numeric data	Pos instr 2, OldStr, <ax>	Searches for “substr” within “string” starting at position “start.” The starting value is optional. If it is missing, MASM begins searching for the string from position one. If MASM cannot find the substring within the string operand, it returns the value zero.
sizestr	string Returns: numeric data	StrSize sizestr OldStr	Returns the size of the string in the operand field.
catstr	string, string, ... Returns: text data	NewStr catstr OldStr, <$$>	Creates a new string by concatenating each of the strings appearing in the operand field of the catstr macro.

Tablica 43: Predefiniowane Makra MASM

Makra substr i catstr zwracają dane tekstowe. Pod wieloma względami są one podobne do dyrektywy textequ ponieważ używamy ich do przydzielenia danej tekstowej do symbolu w czasie asemblacji. Instr i sizestr są podobne do dyrektywy „ = „ o tyle, że zwracają one wartość liczbową.

Makro catstr może wyeliminować potrzebę znalezienia makra MakeLbl w makrze ForLp. Porównajmy następującą wersję ForLp z poprzednia wersją (zobacz „Przykład makra do implementacji pętli for”

```
ForLp           macro       LCV, Start,Stop
                local       ForLoop
                ifndef      $$For&LCV&
$$For&LCV&      =           0
```

```
                        else
$$For&LCV&              =               $$For&LCV&+1
                        endif
                        mov             ax, Start
                        mov             LCV, ax

ForLoop                 textequ         @catstr($for&LCV&,%$$For&LCV&)
&ForLoop&:
                        mov             ax, LCV
                        cmp             ax, Stop
                        jgDone          $$Next&LCV&, %$$For&LCV&
                        endm
```

MASM dostarcza również funkcji makra dla form catstr, instr, sizestr i substr .Dla rozróżnienia tych funkcji makra od odpowiadających predefiniowanych makr. MASM używa nazw @catstr,@instr,@sizestr i @substr. Oto odpowiedniki dla tych operacji:

```
Symbol          catstr          String1, String2,....
Symbol          textequ         @catstr(String1,String2,...)

Symbol          substr          SomeStr, 1, 5
Symbol          textequ         @substr(SomeStr, 1, 5)

Symbol          instr           1, SomeStr,SearchStr
Symbol          =               @substr(1, SomeStr, SearchStr)

Symbol          sizestr         SomeStr
Symbol          =               @sizestr(SomeStr)
```

Name	Parameters	Example
@substr	string, start, length Returns: text data	ifidn @substr(parm, 1, 4), <[bx]>
@instr	start, string, substr Returns: numeric data	if @instr(parm,<bx>)
@sizestr	string Returns: numeric data	byte @sizestr(SomeStr)
@catstr	string, string, ... Returns: text data	jg @catstr($$Next&LCV&, %$$For&LCV&)

Tablica 44: Predefiniowane funkcje makra dla MASMa

Ostatni przykład pokazał jak pozbyć się makr jgDone i jmpLoop w makrze ForLp .Końcowa, poprawiona wersja makr ForLp i Next, wyeliminowała trzy makra i pracująca bez błędów w MASM, może wyglądać tak jak poniżej:

```
ForLp           macro           LCV, Start,Stop
                local           ForLoop

                ifndef`         $$For&LCV&
$$For&LCV&      =               0
                else
$$For&LCV&      =               $$For&LCV& + 1
                endif
                mov             ax, Start
                mov             LCV, ax
ForLoop         textequ         @catstr($For&LCV&, %$$For&LCV&)
&ForLoop&:
                mov             ax, LCV
                cmp             ax, Stop
                jg              @catstr($$Next&LCV&, %$$For&LCV&)
```

```
                endm
Next            macro           LCV
                local           NextLbl
                inc             LCV
                jmp             @catstr($$For&LCV&, %$$For&LCV&)
NextLbl         textequ         @catstr($Next&LCV$, %$$For&LCV&)
&NextLbl&:
                endm
```

MASM dostarcza również dużej liczby wbudowanych zmiennych, które zwracają informacje o bieżącej asemblacji. Poniższa tablica opisuje te wbudowane zmienne czasu asemblacji.

Category	Name	Description	Return result
Date & Time Information	@Date	Returns the date of assembly.	Text value
	@Time	Returns a string denoting the time of assembly.	Text value

Category	Name	Description	Return result
Environment Information	@CPU	Returns a 16 bit value whose bits determine the active processor directive. Specifying the .8086, .186, .286, .386, .486, and .586 directives enable additional instructions in MASM. They also set the corresponding bits in the @cpu variable. Note that MASM sets *all* the bits for the processors it can handle at any one given time. For example, if you use the .386 directive, MASM sets bits zero, one, two, and three in the @cpu variable.	Bit 0 - 8086 instrs permissible. Bit 1 - 80186 instrs permissible. Bit 2 - 80286 instrs permissible. Bit 3- 80386 instrs permissible. Bit 4- 80486 instrs permissible. Bit 5- Pentium instrs permissible. Bit 6- Reserved for 80686 (?). Bit 7- Protected mode instrs okay. Bit 8- 8087 instrs permissible. Bit 10- 80287 instrs permissible. Bit 11- 80386 instrs permissible. (bit 11 is also set for 80486 and Pentium instr sets).
	@Environ	@Environ(name) returns the text associated with DOS environment variable name. The parameter must be a text value that evaluates to a valid DOS environment variable name.	Text value
	@Interface	Returns a numeric value denoting the current language type in use. Note that this information is similar to that provided by the opattr attribute. The H.O. bit determines if you are assembling code for MS-DOS/Windows or OS/2. This directive is mainly useful for those using MASM's simplified segment directives. Since this text does not deal with the simplified directives, further discussion of this variable is unwarranted.	Bits 0-2 000- No language type 001- C 010- SYSCALL 011- STDCALL 100- Pascal 101- FORTRAN 110- BASIC Bit 7 0- MS-DOS or Windows 1- OS/2
	@Version	Returns a numeric value that is the current MASM version number multiplied by 100. For example, MASM 6.11's @version variable returns 611.	Numeric value
File Information	@FileCur	Returns the current source or include file name, including any necessary pathname information.	Text value
	@File-Name	Returns the current source file name (base name only, no path information). If in an include file, this variable returns the name of the source file that included the current file.	Text value
	@Line	Returns the current line number in the source file.	Numeric value

Category	Name	Description	Return result
Segment [a] Information	@code	Returns the name of the current code segment.	Text value
	@data	Returns the name of the current data segment.	Text value
	@FarData?	Returns the name of the current far data segment.	Text value
	@Word-Size	Returns two if this is a 16 bit segment, four if this is a 32 bit segment.	Numeric value
	@Code-Size	Returns zero for Tiny, Small, Compact, and Flat models. Returns one for Medium, Large, and Huge models.	Numeric value
	@DataSize	Returns zero for Tiny, Small, Medium, and Flat memory models. Returns one for Compact and Large models. Returns two for Huge model programs.	Numeric value
	@Model	Returns one for Tiny model, two for Small model, three for Compact model, four for Medium model, five for Large model, six for Huge model, and seven for Flag model.	Numeric value
	@CurSeg	Returns the name of the current code segment.	Text value
	@stack	The name of the current stack segment.	Text value

Tablica 45:Przedefiniowane zmienne czasu asemblacji MASMA

Chociaż jest niewystarczająco miejsca do omówienia w szczegółach o możliwych zastosowaniach dla każdej z tych zmiennych, kilka przykładów można zademonstrować jako możliwości. Inne zastosowania tych zmiennych będą pojawiały się w całym tekście, jednak największe wrażenie zrobi samodzielne ich odkrywanie.

Zmienna @CPU jest całkiem użyteczna jeśli chcemy zasemblować różne sekwencje kodu w programie dla różnych procesorów. Sekcja o asemblacji warunkowej w tym rozdziale opisuje jak można stworzyć symbol określający czy assemblujemy kod dla procesora 80386 lub późniejszych lub procesora 8086.Symbol @CPU dostarcza nam symbolu ,który mówi dokładnie jakie instrukcje są dozwolone w danym punkcie naszego programu. Poniżej jest przedstawiony przykład zastosowania zmiennej @CPU:

```
if      @CPUand 100b    ;Potrzebujemy procesora 80286 lub późniejszych
shl     ax, 4           ;dla tych instrukcji
else                    ;musi być procesor 8086
mov     cl, 4
shl     ax, cl
endif
```

Możemy użyć dyrektywy @Line do wprowadzenia specjalnej wiadomości diagnostycznej w naszym kodzie. następujący kod wydrukuje informację o błędzie wliczając w to numer linii w pliku źródłowym naruszonego twierdzenia, jeśli wykryje błąd w czasie wykonania:

```
mov     ax, ErrorFlag
cmp     ax, 0
je      NoError
mov     ax, @Line       ;ładuje AX z bieżącej linii #
call    Printerror      ;drukuje info o błędzie i Linię #
jmp     Quit            ;Kończy program
```

8.14.9 MAKRA A PRZYRÓWNYWANIE TEKSTU

Marka, funkcje makra i przyrównywanie tekstu wszystkie zastępują tekst w programie. Chociaż jest jakieś podobieństwo miedzy nimi, w rzeczywistości służą one różnym celom w programie języka asemblera.

Przyrównywanie tekstu wykonuje zastąpienie pojedynczego tekstu w linii. Nie pozwala na żadne parametry. Jednakże, możemy zastąpić tekst gdziekolwiek w linii przyrównaniem tekstu. Możemy rozszerzyć przyrównanie tekstu na etykietę, mnemonik, operand lub nawet pole komentarza. Ponadto możemy zastąpić pola wielokrotne, nawet w całej linii pojedynczym symbolem.

Funkcje makra są poprawne w tylko w polu operandu. Jednak możemy przekazać parametry do funkcji makra czyniąc je bardziej ogólnym niż po prostu przyrównaniem tekstu.

Makra proceduralne pozwalają nam emitować sekwencje instrukcji (z przyrównywaniem tekstu**,** możemy emitować ,najwyżej, jedną instrukcję).

8.14.10 MAKRA: DOBRE I ZŁE WIEŚCI

Makra oferują znaczną wygodę. Pozwalają nam na wprowadzenie kilku instrukcji do naszego pliku źródłowego poprzez wpisanie pojedynczej komendy. Może to zaoszczędzić niewiarygodną ilość pisaniny kiedy wprowadzamy ogromną tablicę, której każda linia zawiera jakieś dziwaczne ale powtarzalne obliczenia. .Jest użyteczne (w pewnych przypadkach) dla wspomożenia uczynienia naszych programów bardziej czytelnymi.. Niewielu będzie się sprzeczało, że ForLp I,1 ,10 nie jest bardziej czytelne niż odpowiadający kod 8086.Niestety,jest łatwo stworzyć kod, który jest niewydolny, trudny do czytania i trudny do pielęgnacji.

Wielu tak zwanych „zaawansowanych" programistów assemblerowych poniósł pomysł, że mogą tworzyć swoje własne instrukcje przez definicję makra i zaczynają tworzyć makra dla każdej wyobrażalnej funkcji pod słońcem. Makro COPY przedstawione wcześniej jest dobrym przykłądem.8086 nie wspiera operacji przesunięcia z pamięci do pamięci. Spoko, stworzymy makro, które zrobi tę robotę dla nas. Wkrótce program asemblerowy nie będzie wcale wyglądał jak asembler 8086.Zamiast tego będzie duża liczba instrukcji będących wywołaniami makra. Być może teraz jest dobrze programiście który stworzył wszystkie te makra i gruntownie zrozumiał ich działanie. Programista 8086,który nie jest zaznajomiony z tymi makrami jest to wszystko bzdurą .Utrzymywanie programu ,który ktoś napisał, który zawiera „nowe" instrukcje implementowane przez makro, jest zadaniem strasznym .Dlatego też ,powinniśmy rzadko stosować makra jako środka do tworzenia nowych instrukcji na 80x86

Inny problem z makrami, to to, że mają one tendencję do ukrywania skutków ubocznych .Rozważmy makro COPY prezentowane wcześniej. Jeśli napotkamy instrukcję w postaci COPY VAR1, VAR2 w programie asemblerowym, będziemy myśleć, że jest to niewinna instrukcja, która kopiuje VAR2 do VAR1.Błąd!Niszczy ona również zawartość rejestru ax pozostawiając kopię wartości VAR2 w rejestrze ax. Wywołanie tego makra nie czyni tego wyraźnie. Rozważmy taką sekwencję kodu:

```
mov     ax, 5
copy    Var2, Var1
mov     Var1, ax
```

Ta sekwencja kodu kopiuje Var1 do Var2 a potem (podobno) przechowuje pięć w Var1.Niestety,makro COPY zmiotło wartość w ax (pozostawiając samą pierwotną wartość zawartą w Var1),więc sekwencja instrukcji nie modyfikuje wcale Var1!

Innym problemem z makrami jest wydajność. Rozważmy następujące wywołania makra COPY:

```
copy    Var3, Var1
copy    Var2, Var1
copy    Var0, Var1
```

Te trzy instrukcje generują kod:

```
mov     ax, Var1
mov     Var3, ax
mov     ax, Var1
mov     Var2, ax
mov     ax, Var1
mov     Var0,  ax
```

wyraźnie widać ,że ostatnie dwie instrukcje mov ax, Var1 są zbyteczne. rejestr ax już zawiera kopie Var1,więc nie musimy ponownie ładować ax tą wartością. Niestety, ta niedogodność jest zupełnie oczywista w kodzie rozszerzonym ,nie jest oczywista wcale w wywołaniu makra.

Innym problemem z makrami jest złożoność. Żeby wygenerować wydajny kod, musimy stworzyć niezmiernie złożone makro używając asemblacji warunkowej (zwłaszcza ifb, ifidn ,itp.),pętli reapet (opisanej trochę później) i innych dyrektyw. Niestety ,makra te są małymi programikami same w sobie. Możemy mieć błędy w naszych makrach, kiedy mamy błąd w naszym programie asemblerowym. Przez większa złożoność naszych makr staje się bardziej prawdopodobne, że zawarte w nich błędy staną się błędami w naszych programach, kiedy wywołamy makro.

Nadużywanie makr, zwłaszcza złożonych, tworzy trudny do odczytania kod ,który jest trudny do pielęgnacji. Mimo entuzjastycznych zapewnień, tych, którzy kochają makra ,niepohamowane stosowanie makr wewnątrz programu generalnie powoduje więcej błędów niż pomaga. Jeśli będziemy stosować makra, bądźmy ostrożni.

Jednak jest i dobra strona makr. Jeśli ujednolicimy zbiór makr i udokumentujemy wszystkie nasze programy przez stosowanie makr, mogą one nam pomóc uczynić nasze programy bardziej czytelnymi. Zwłaszcza jeśli te makra mają łatwo identyfikowalne nazwy. Standardowa Biblioteka UCR dla Programistów Języka Assemblera 80x86 stosuje makra dla większości wywołań biblioteki. Przeczytamy więcej o Standardowej Bibliotece UCR w następnym rozdziale.

8.15 OPERACJE REPEAT

Innym formatem makra (przynajmniej zdefiniowanym przez Microsoft) jest makro repeat. Makro repeat jest niczym więcej niż pętlą, która powtarza instrukcje wewnątrz pętli określona ilość razy .Są trzy typy makra repeat dostarczane przez MASM: repeat/rept.for/irp i forc/irpc .Makro repeat/rept używa następującej składni:

```
        repeat      wyrażenie
    <instrukcje>
        endm
```

Wyrażenie musi być wyrażeniem liczbowym, które oblicza stałą bez znaku. Dyrektywa repeat powiela wszystkie instrukcje pomiędzy repeat a endm wiele razy. Poniższy kod generuje tablicę 26 bajtową, zawierającą 26 dużych znaków:

```
ASCIICode       =           'A'
                repeat      26
                byte        ASCIICode
ASCIICode       =           ASCIICode+1
                endm
```

Symbol ASCIICode jest powiązany z kodem ASCII dla 'A'. Pętla powtarza się 26 razy, za każdym razem emitując bajt z wartością ASCIICode. Również jest zwiększany symbol ASCIICode po każdej powtórce ,tak aby zawierał kod ASCII następnego znaku w tabeli ASCII. Faktycznie są generowane następujące instrukcje:

```
            byte        'A'
            byte        'B'
            -
            -
            -
            byte        'Y'
            byte        'Z'
ASCIICode   =           27
```

Zauważmy, że pętla repeat wykonuje się w czasie asemblacji a nie w czasie wykonania. Repeat nie jest mechanizmem dla tworzenia pętli wewnątrz naszych programów; używamy jej dla replikowania części kodu wewnątrz programu. Jeśli chcemy stworzyć pętlę, która wykonuje się pewną liczbę razy wewnątrz programu, użyjemy instrukcji loop. Chociaż poniższe dwie sekwencje kodu dają ten sam wynik, nie są one takie same:

```
;sekwencja kodu używająca pętli czasu wykonania:
            mov         cx, 10
AddLp:      add         ax, [bx]
            add         bx, 2
            loop        AddLp
;Sekwencja kodu używająca pętli czasu asemblacji:
            repeat      10
            add         ax, [bx]
            add         bx, 2
            endm
```

Pierwsza, powyższa sekwencja kodu emituje cztery instrukcje maszynowe do pliku kodu wynikowego. W czasie asemblacji, CPU 80x86 wykonują instrukcje pomiędzy AddLp a instrukcją loop dziesięć razy pod kontrolą instrukcji loop. Druga sekwencja kodu emituje 20 instrukcji do pliku kodu wynikowego. W czasie wykonania, CPU 80x86 po prostu wykonuje te 20 instrukcji sekwencyjnie ,bez żadnej transmisji sterowania. Druga forma będzie szybsza ponieważ 80x86 nie musi wykonywać instrukcji loop co trzecią instrukcję. Z drugiej strony, druga wersja jest również dużo większa ponieważ replikuje treść pętli dziesięć razy w pliku kodu wynikowego.

W odróżnieniu od makr standardowych, nie definiujemy i wywołujemy makr repeat oddzielnie. MASM emituje kod pomiędzy dyrektywami repeat i endm po napotkaniu dyrektywy repeat. Nie jest to oddzielna faza wywołania. Jeśli chcemy stworzyć makro repeat, które może być wywoływane w całym naszym programie, rozważmy cos takiego:

```
REPTMakro       macro       Cout
                repeat      Cout
              <instrukcje>
                endm
                endm
```

Poprzez umieszczenie makra repeat wewnątrz makra standardowego, możemy wywołać makro repeat gdziekolwiek w naszym programie poprzez wywołanie makra REPTMakro. Zauważmy, że potrzebujemy dwóch dyrektyw endm ,jedną do zakończenia makra repeat, jedną do zakończenia makra standardowego.

Rept jest synonimem dla repeat .Repeat jest nowszą forma, MASM wspiera rept dla kompatybilności ze starszymi plikami źródłowymi. Zawsze powinniśmy używać formy repeat.

8.16 MAKRO OPERACJE FOR I FORC

Inną postacią makra repeat jest makro for. Makro to przybiera następującą postać:

```
        for             PARAMETR, <ITEM1{ITEM2{ITEM3{...}}}>
<instrukcje>
        endm
```

Nawiasy ostre są wymagane wokół pozycji w polu operandu dyrektywy for. Nawiasy otaczają opcjonalne pozycje ,nawiasy nie powinny się pojawiać w polu operandu.

Dyrektywa for replikuje instrukcje pomiędzy for i endm raz dla każdej pozycji pojawiającej się w polu operandu. Co więcej dla każdej iteracji pierwszemu symbolowi w polu operandu jest przypisywana wartość kolejnej pozycji drugiego parametru. Rozważmy następującą pętle:

```
        for             value, <0,1,2,3,4,5>
        byte            value
        endm
```

Pętla ta emituje sześć bajtów zawierających wartości zero, jeden ,dwa....pięć. Jest to absolutnie identyczne z sekwencją instrukcji

```
        byte            0
        byte            1
        byte            2
        byte            3
        byte            4
        byte            5
```

Pamiętajmy, że pętla for, podobnie jak pętla repeat wykonuje się w czasie asemblacji a nie w czasie wykonania.

Drugi operand for nie musi być stałą tekstową; możemy dostarczyć parametr makra, wynik funkcji makra lub przyrównywanie tekstowe do tej wartości. Zapamiętajmy, mimo że ten parametr musi być rozszerzony do wartości tekstowej z ogranicznikami tekstu wokół niej.

Irp jest starym ,przestarzałym, synonimem dla for. MASM pozostawił irp dla kompatybilności ze starszymi plikami źródłowymi. Jednak zawsze powinniśmy stosować dyrektywę for.

Trzecią formą makra pętli jest makro forc .Różni się od makra for w tym, że powtarza pętlę ilość razy wyszczególnioną przez długość ciągu znaków zamiast przez liczbę przedstawionych operandów. Składnia dla dyrektywy forc:

```
        forc            parametr, <string>
<instrukcje>
        endm
```

Instrukcje w pętli są powtarzane raz dla każdego znaku w operandzie ciągu. Nawiasy ostre muszą pojawić się wokół ciągu. Rozważmy następującą pętlę:

```
        forc            value, <012345>
        byte            value
        endm
```

Pętla ta tworzy taki sam kod jak przykład dla powyższej dyrektywy for.

Irpc jest starym synonimem dla forc dostarczanym dla kompatybilności. Jednak zawsze powinniśmy używać forc w naszych kodach.

8.17 OPERACJE MAKRA WHILE

Makro while pozwala nam powtarzać sekwencję kodu w naszym pliku asemblerowym nieokreśloną ilość razy .Czas asemblacji wyrażenia, które while oblicza przed emitowaniem kodu dla każdej pętli, determinuje czy ją powtarza. Składnia dla tego makra:

```
        while           wyrażenie
<instrukcje>
        endm
```

Makro to oblicza wyrażenie w czasie asemblacji; jeśli wartość tego wyrażenia jest zero, makro while ignoruje instrukcje aż do odpowiadającej mu dyrektywy endm. Jeśli obliczone wyrażenie jest nie zerowe (prawda) wtedy MASM assembluje instrukcje do dyrektywy endm i oblicza ponownie wyrażenie aby zobaczyć czy powinno assemblować treść pętli while ponownie.

Zwykle ,dyrektywa while powtarza instrukcje pomiędzy while i endm tak długo jak obliczone wyrażenie jest prawdą. Jednakże, możemy również użyć dyrektywy exitm do wcześniejszego zakończenia rozszerzania się treści loop .Zapamiętajmy, że musimy dostarczyć jakiś warunek, który kończy pętlę ,w

przeciwnym wypadku, MASM będzie kontynuował pętlę nieskończoną i emitował kod do pliku kodu wynikowego dopóki dysk się nie zapełni (lub po prostu będzie wykonywał pętlę nieskończoną gdy pętla nie emituje żadnego kodu).

8.18 PARAMETRY MAKRA

Standardowe makra MASMa są bardzo elastyczne. Jeśli liczba rzeczywistych parametrów (tych obecnych w polu operandu wywołania makra) nie odpowiada liczbie parametrów formalnych (tych pojawiających się w polu operandu definicji makra),MASM nie koniecznie będzie narzekał. jeśli jest więcej parametrów rzeczywistych niż formalnych .MASM zignoruje te parametry dodatkowe i wygeneruje ostrzeżenie. Jeśli będzie więcej parametrów formalnych niż rzeczywistych zastąpi pusty ciąg („<>") dla dodatkowych parametrów Poprzez zastosowanie dyrektyw asemblacji warunkowej ifb i ifnb, możemy przetestować te ostatnie warunki. Podczas gdy ta technika zastępowania parametrów jest elastyczna pozostawia ona otwarte możliwości błędów. Jeśli chcemy, żeby programista dostarczył dokładnie trzy parametry ,a dostarczył mniej, MASM nie będzie generował błędu. jeśli zapomnimy przetestować obecność każdego parametru używając ifb, możemy wygenerować zły kod. Do przezwyciężenia tego ograniczenia MASM dostarczył również zdolność do wyszczególniania tych pewnych parametrów makr, które są wymagane. Możemy również przydzielić wartość domyślną do parametru jeśli jest to wymagane. W końcu MASM dostarcza również możliwości pozwalających na zmienne liczbowe argumentów makra.

Jeśli wymagamy aby programista dostarczył szczególnych parametrów makra, po prostu dopisujemy „:req" po parametrze makra w definicji makra. W czasie asemblacji MASM wygeneruje błąd jeśli to szczególne makro zaginie.

```
Needs2Param     macro       param1:req, param2:req
                -
                -
                -
                endm
                -
                -
                -
                Needs2Param  ax          ;generuje błąd
                Needs2Param              ;generuje błąd
                Needs2Param  ax,  bx     ;pracuje dobrze
```

Inną możliwością jest mieć makro dostarczające wartość domyślną do makra jeśli zaginął on z listy parametrów rzeczywistych robiąc to po prostu używamy operatora „:=text>" bezpośrednio po nazwie parametru w liście parametrów formalnych .Na przykład, funkcja BIOS int 10h dostarcza różnych usług video. Jednym z bardziej powszechnych zastosowań usług video jest funkcja ah=0eh,która wyprowadza znak z al. Na wyświetlacz.

Poniższe makro pozwala wywołać tą funkcję której chcemy użyć, i domyślnie funkcje 0eh jeśli nie wyszczególnimy parametru:

```
Video       macro       service := <0eh>
            mov         ah, service
            int         10h
            endm
```

Ostatnia cech makr MASMa wspierana jest zdolnością do przetwarzania parametrów jako zmiennych liczbowych. Robiąc to, po prostu umieszczamy operator „:vararg" po ostatnim formalnym parametrze w liście parametrów. MASM wiąże pierwsze n parametrów rzeczywistych z odpowiadającymi parametrami formalnymi pojawiającymi się przed argumentem zmiennej, tworząc potem przyrównanie tekstu wszystkich pozostałych parametrów do parametru formalnego z przyrostkiem operatora „:vararg". Możemy zastosować makro for do wydzielenia każdego parametru z tej listy argumentów zmiennej. Na przykład poniższe makro pozwala nam zadeklarować przypadkową liczbę dwuwymiarowych tablic, wszystkie o tych samych rozmiarach. Pierwsze dwa parametry wyszczególniają liczbę wierszy i kolumn, pozostałe opcjonalne parametry wyszczególniają nazwę tych tablic:

```
MkArrays        macro       Numrow:req,NumCols:req,Names:vararg
                for         AryName,Names
                word        NumRows  dup  (Numcols  dup  (?))
                endm
                endm
                -
                -
```

-
MkArrays 8, 12, a, b, x, y

8.19 KONTROLA LISTINGU

MASM dostarcza kilku dyrektyw asemblerowych ,które są użyteczne przy kontroli asemblera. Dyrektywy te to echo ,%out, title, subttl ,page, .list, .nolist i .xlist. jest kilka innych, ale te są najważniejsze

---

8.19.1 DYREKTYWY ECHO I %OUT

Dyrektywy echo i %out po prostu drukują cokolwiek pojawi się w ich polu operandu na wyświetlaczu podczas asemblacji. Pewne przykłady echo i %out pojawiły się w sekcji o asemblacji warunkowej i makrach. Zauważmy, że %out jest starszą postacią echo dostarczoną dla kompatybilności ze starymi kodami źródłowymi. Powinniśmy używać echo we wszystkich nowych kodach.

8.19.2 DYREKTYWA TITLE

Dyrektywa asemblera title przydziela tytuł do naszego pliku źródłowego. Tylko jedna dyrektywa title może pojawić się w naszym programie. Składani tej dyrektywy to:

title tekst

MASM będzie drukował wyszczególniony tekst na górze każdej strony listingu asemblera

---

8.19.3 DYEREKTYWA SUBTTL

Dyrektywa subttl jest podobna do dyrektywy title, z z wyjątkiem wielu napisów pojawiających się wewnątrz naszego pliku źródłowego. podtytuły pojawiają się bezpośrednio poniżej tytułu na górze każdej strony w listingu asemblera. składnia dla dyrektywy subttl :

Subttl tekst

Wyszczególniony tekst stanie się nowym podtytułem. Zauważmy ,że MASM nie wydrukuje nowego podtytułu aż do pojawienia się nowej strony. Jeśli życzymy sobie umieścić podtytuł na tej samej stronie na jakiej bezpośrednio następuje kod po dyrektywie, użyjemy dyrektywy page (omówionej poniżej) do wymuszenia pojawienia się strony

---

8.19.4 DYREKTYWA PAGE

Dyrektywa page spełnia dwie funkcje – może wymusić pojawianie się strony w asemblowanym listingu i może ustawić szerokość i długość na urządzeniu wyjściowym .Do wymuszenia pojawienia strony używamy następującej formy dyrektywy page:

Page

Jeśli umieścimy znak plus ,"+",w polu operandu, wtedy MASM wykona przerwanie, zwiększenie numeru sekcji i przestawienie numeru strony o jeden. MASM drukuje liczbę stron używając formatu

Section-page

Jeśli chcemy wykorzystać udogodnienie numeru sekcji, będziemy musieli ręcznie wprowadzić przerwanie strony (z operandem „+") przed każdą nową sekcja

Druga forma polecenia page pozwala nam ustawić wartość długości i szerokości wydruku strony. Przyjmuje postać:

page length, width

gdzie długość jest liczbą linii na stronę (domyślnie 50,ale 50-60 jest lepszym wyborem dla większości drukarek) a szerokość jest liczbą znaków na linię. Domyślna szerokość to 80 znaków .Jeśli nasza drukarka jest zdolna do drukowania 132 kolumn, powinniśmy zmienić tą wartość na 132,więc nasz listing będzie łatwiejszy do czytania. Zauważmy, że niektóre drukarki ,nawet jeśli karetka jest szeroka na 8-1/2" ,wydrukujemy przynajmniej 132 kolumny w trybie zagęszczonym. Typowo jakiś znak sterujący musi być wysłany do drukarki i umieszczony w trybie zagęszczonym. Możemy wprowadzić taki znak sterujący w komentarzu na początku naszego listingu źródłowego.

8.19.5 DYREKTYWY .LIST, .NOLIST I .XLIST

Dyrektywy .list, .nolist i .xlist mogą być używane do wybranej części listy naszego pliku źródłowego podczas asemblacji.. List włącza listing, .Nolist wyłącza listing. .Xlist jest przestarzałą formą .Nolist dla starszych kodów.

Poprzez dodanie szczypty tych trzech dyrektyw w całym naszym pliku źródłowym, możemy listować tylko te sekcje kodu, które nas interesują. Żadna z tych dyrektyw nie akceptuje żadnego operandu Przybierają następujące formy:

.list
.nolist
.xlist

8.19.6 INNE DYREKTYWY LISTINGU

MASM dostarcza kilku innych dyrektyw kontrolnych listingu, których ten rozdział nie omawia. Pozwalają nam kontrolować makra, segmenty asemblacji warunkowej i inne w pliku listingu. Proszę zobaczyć dodatki po więcej szczegółów o tych dyrektywach.

8.20 ZARZĄDZANIE DUŻYMI PROGRAMAMI

Większość programów asemblerowych nie jest zupełnie autonomicznymi programami. Generalnie rzecz biorąc, wywołujemy różne biblioteki standardowe lub inne podprogramy które nie są zdefiniowane w naszym głównym programie. Na przykład, prawdopodobnie zauważyliśmy, że 80x86 nie dostarcza żadnych instrukcji takich jak „read", "write" lub „printf" dla wykonania operacji I/O .Faktycznie, jedynymi instrukcjami dla I/O, jakie zawiera 80x86 są instrukcje in i out, które są w rzeczywistości specjalnymi instrukcjami mov, i dyrektywy echo /%out, które wykonują się w czasie asemblacji, a nie jak chcemy w czasie wykonania. Czy nie ma żadnego sposobu wykonania I/O w asemblerze? Oczywiście jest. Możemy napisać procedury, które wykonują operacje I/O takie jak „read", „write". Niestety napisanie takich podprogramów jest zadaniem złożonym ,a początkujący programiści asemblerowi nie są przygotowani do takich zadań. Wtedy z pomocą przychodzi Standardowa Biblioteka UCR dla Programistów Języka Asemblera Dla 80x86.Są to upakowane procedury, które możemy wywoływać do wykonania prostych operacji I/O, takich jak „printf"

Standardowa Biblioteka UCR zawiera tysiące linii kodu źródłowego. Wyobraźmy sobie jak trudne byłoby programowaniem gdybyśmy musieli włączyć te tysiące linii kodu do naszych prostych programów. Na szczęście nie musimy.

Dla małych programów, pracujących z pojedynczym plikiem źródłowym wszystko jest dobrze. Dla dużych programów staje się to nieporęczne (rozpatrzmy powyższy przykład musząc wprowadzić cała Bibliotekę UCR do każdego naszego programu).Co więcej, kiedy zdebaggujemy i przetestujemy dużą część naszego kodu, kontynuacja asemblacji tego samego kodu, kiedy zrobimy małą zmianę jakiejś innej części naszego kodu jest stratą czasu. Na przykład Standardowej Bibliotece UCR asemblacja zajmuje kilka minut, nawet na szybkich maszynach. Wyobraźmy sobie ,że musimy czekać pięć lub dziesięć minut na szybkim Pentium na asemblację programu, w którym zmieniliśmy jedną linijkę!

Podobnie jak w HLL'ach, rozwiązaniem jest oddzielna kompilacja (lub oddzielna asemblacja w przypadku MASMa).Po pierwsze, rozkładamy nasze duże pliki źródłowe na wykonywalne kawałki. Potem assemblujemy oddzielne pliki do modułów kodu wynikowego. W końcu, linkujemy moduły wynikowe razem do postaci kompletnego programu. Jeśli musimy dokonać małej zmiany w jednym z modułów, musimy tylko zreassemblować jeden z modułów, nie musimy reasemblować całego programu.

Standardowa Biblioteka UCR pracuje w ten sposób dokładniej. Biblioteka Standardowa jest już zasemblowana i gotowa do użycia. po prostu wywołujemy podprogram z Biblioteki Standardowej i linkujemy nasz kod z Biblioteką Standardową używając programu linkera. Oszczędza to ogromną ilość czasu kiedy wykorzystujemy program używający kodu Biblioteki Standardowej. Oczywiście ,możemy łatwo stworzyć własny moduły wynikowe i zlinkować je razem z naszym kodem. Możemy nawet dodawać nowe podprogramy do biblioteki Standardowej aby były dostępne do zastosowania w przyszłych programach, które napiszemy w przyszłości.

„Programming in the large" jest terminem inżynierów oprogramowania ukutym na potrzeby opisania procesów, metodologii i narzędzi przy rozwoju dużych projektów programistycznych. W zasadzie każdy ma swój własny pomysł co to jest „duży", oddzielna kompilacja, i jakaś konwencja stosowania oddzielnej kompilacji, są jednym z większych technik „programming in large". Poniższa sekcja opisuje narzędzia MASMa dla oddzielnej kompilacji i jak wydajniej stosować te narzędzia w naszych programach.

8.20.1 DYREKTYWA INCLUDE

Dyrektywa include. kiedy znajduje się w pliku źródłowym, przełącza program z pliku bieżącego na plik wyszczególniony w liście parametrów include. Pozwala to nam na zbudowanie pliku tekstowego zawierającego identyfikatory ,makra, kod źródłowy i inne pozycje asemblerowe, i plik nagłówkowy do asemblacji kilku oddzielnych programów. Składnia dyrektywy include

```
include        nazwa pliku
```

Nazwa pliku musi być poprawną nazwą DOS'a. MASM łączy wyszczególniony plik do asemblacji w punkcie dyrektywy include .Zauważmy, że możemy zagnieżdżać instrukcje include wewnątrz plików,. To znaczy, plik zawarty wewnątrz innego pliku podczas asemblacji może wywołać trzeci plik.

Stosowanie dyrektywy include prze nią samą nie dostarcza oddzielnej kompilacji. Możemy użyć dyrektywy include do podzielenia dużego pliku na oddzielne moduły i łączyć te moduły razem, kiedy assemblujemy nasz plik. Poniższy przykład łączy pliki PRINTF.ASM i PUTC.ASM podczas asemblacji naszego programu:

```
include        printf.asm
include        putc.asm
```

<Kod dla naszego programu >
end

Teraz nasz program będzie przynosił korzyść z modularności. Niestety ,nie oszczędzimy na czasie konstruowania programu. Dyrektywa include wkłada plik źródłowy w punkcie include podczas asemblacji, dokładnie jak gdybyśmy napisali ten kod sami. MASM musi jeszcze zasemblować ten kod a to zabiera czas. Gdybyśmy włożyli wszystkie pliki podprogramów Biblioteki Standardowej ,nasza asemblacja trwałaby wiecznie.

Generalnie, nie powinniśmy używać dyrektywy include do włączania pliku źródłowego jak pokazano powyżej. zamiast tego powinniśmy używać dyrektywy include do wprowadzania zbioru stałych, makr, deklaracji zewnętrznych procedur, i innych takich pozycji do programu. Typowo, asembler zawiera pliki, nie zawierające żadnego kodu maszynowego (na zewnątrz makr).Cel stosowania plików include w ten sposób, stanie się jaśniejszy po zobaczeniu jak pracują deklaracje public i external.

8.20.2 DYREKTYWY PUBLIC,EXTERN I EXTRN

Technicznie, dyrektywa include dostarcza nam wszystkich udogodnień potrzebnych do stworzenia programów modularnych .Możemy zgromadzić bibliotekę modułów, każda zawierająca jakiś specyficzny podprogram i zawrzeć niezbędne moduły do programu asemblera używając odpowiedniej komendy include. MASM (i towarzyszący mu program LINK) dostarczają lepszego sposobu: symboli external i public.

Jeden ważny problem z mechanizmem include jest to ,że zdebuggowanie podprogramu, wprowadzonego do asemblacji jest marnotrawstwem czasu ponieważ MASM musi zreassemblować wolny od błędów kod za każdym razem ,kiedy asembujemy główny program. Dużo lepszym rozwiązaniem będzie wcześniejsza asemblacja zdebuggowanego modułu i zlinkowanie razem z modułem wynikowym zamiast reasemblacja całego programu za każdym razem ,kiedy zmieniamy pojedynczy moduł. To jest to czego dostarczają dyrektywy public i extern. Extrn jest starszą dyrektywa, która jest synonimem extern. Dołączona jest dla kompatybilności ze starszymi plikami źródłowymi.. powinniśmy zawsze stosować dyrektywę extern w nowym kodzie źródłowym.

Aby skorzystać z public i extern musimy stworzyć przynajmniej dwa pliki źródłowe .Jeden plik zawiera zbiór zmiennych i procedur używanych przez drugi. Drugi plik używa tych zmiennych i procedur bez wiedzy jak są one implementowane. Dla zademonstrowania rozważymy następujące dwa moduły:

```
;Module #1:
                public          Var1,Var2,Proc1
DSEG            segment         para public  'data'
Var1            word            ?
Var2            word            ?
DSEG            ends

CSEG            segment         para public  'code'
                assume          cs:cseg, ds.:dseg
Proc1           proc            near
                mov             ax, Var1
                add             ax, var2
                mov             Var1, ax
                ret
Proc1           endp
CSEG            ends
                End
;Module #2:
                extern          Var1:word,Var2:word, Proc1:near
CSEG            segment         para  public  'code'
                -
                -
                -
                mov             Var1, 2
                mov             Var2, 3
                call            Proc1
                -
                -
                -
CSEG            ends
```

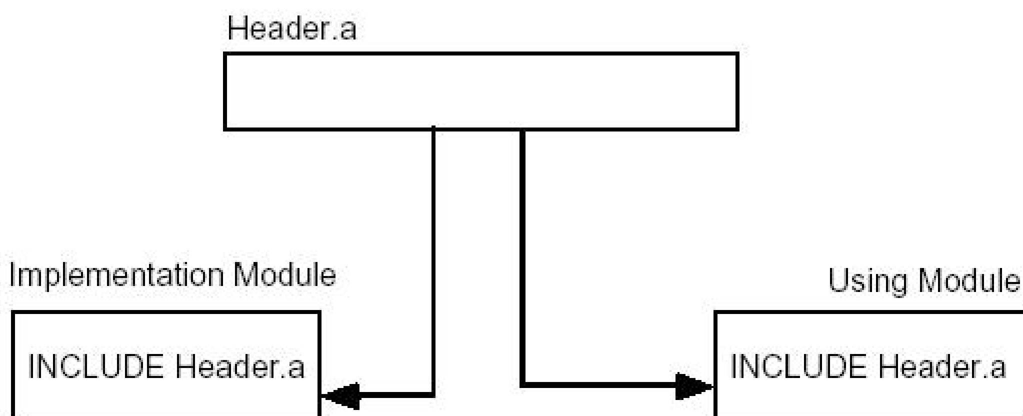
end

Module #2 odnosi się do Var1,Var2 i Proc1,ale mimo to symbole te są zewnętrzne dla modułu #2. Dlatego też musimy zadeklarować je dyrektywa extern. Dyrektywa ta przyjmuje postać:

Extern nazwa: typ {nazwa:typ...}

Nazwa jest nazwą symbolu zewnętrznego, a typ jest typem tego symbolu. Typ może być near, far, proc, byte, word, dword, qword ,tbyte, abs (wartość bezwzględna, która jest stałą),lub innym zdefiniowanym typem przez użytkownika.

Bieżący moduł używa tego typu deklaracji. Ani MASM ani linker nie sprawdzają deklaracji typu w stosunku do modułu definiującego nazwę aby zobaczyć czy typ jest zgodny. Dlatego też musimy korzystać ostrożnie kiedy definiujemy symbole zewnętrzne. Dyrektywa public pozwala nam eksportować wartość symbolu do modułów zewnętrznych. Deklaracja przybiera postać:



Rysunek 8.8: Użycie pojedynczego pliku Include do implementacji i stosowania modułów

Public name {name...}

Każdy symbol pojawiający się w polu operandu instrukcji public jest dostępny jako symbol zewnętrzny do innego modułu. Podobnie, wszystkie zewnętrzne symbole wewnątrz modułu muszą pojawić się wewnątrz instrukcji public w jakimś innym module.

Kiedy stworzymy moduł źródłowy ,powinniśmy najpierw zasemblować plik zawierający deklaracje public. Z MASMem 6.x będziemy stosować komendę taka jak

ML/c.pubs.asm

Opcja „/c" mówi MASMowi aby wykonał asemblację „tylko kompilacja" To znaczy ,nie będzie próbował linkować kodu po skutecznej asemblacji. Tworzy moduł wynikowy „pubs.obj"

Następnie asembujemy plik zawierający definicje zewnętrzne i linkujemy kod stosując komendę MASMa:

ML exts.asm pubs.obj

Zakładając, że nie ma żadnych błędów, stworzymy plik „exts.exe", który jest zlinkowaną i wykonywalną postacią programu.

Zauważmy, ze dyrektywa extern definiuje symbol w naszym pliku źródłowym. Każda próba przedefiniowania symbolu gdzieś w programie stworzy błąd „symbolu powtórzonego" To, okazuje się, jest źródłem problemów, które Microsoft rozwiązuje dyrektywa externdef

8.20.3 DYREKTYWA EXTERNDEF

Dyrektywa externdef jest kombinacją public i extern połączonych w jedno. Używa tej samej składni jak dyrektywa extern to znaczy umieszczamy całą listę nazwa: typ w polu operandu. Jeśli MASM nie napotka innej definicji symbolu w bieżącym pliku źródłowym, externdef będzie się zachowywać dokładnie jak instrukcja extern. Jeśli symbol nie pojawi się w pliku źródłowym, wtedy externdef zachowuje się jak komenda public. Z dyrektywą externdef rzeczywiście nie musimy stosować instrukcji public lub extern, chyba, że czujemy się jakoś zmuszeni do zrobienia tego.

Wielką korzyścią dyrektywy externdef jest to ,że pozwala nam minimalizować powielanie wysiłku w naszym pliku źródłowym .Przypuśćmy na przykład, że chcemy stworzyć moduł z grupą wspomagających podprogramów dla innych programów. Oprócz współdzielenia jakichś podprogramów i jakichś zmiennych, przypuśćmy chcemy również dzielić stałe i makra.. Mechanizm pliku include dostarcza doskonałego sposobu do uczynienia tego. po prostu tworzymy plik include zawierający stałe, makra i definicje externdef i zawieramy ten plik w module, który implementuje nasze podprogramy i w module, który używa tych podprogramów (zobacz rysunek 8.8)

Zauważmy, że extern i public nie działają w tym przypadku ponieważ implementacja modułu potrzebuje dyrektywy public a zastosowanie modułu potrzebuje dyrektywy extern. będziemy musieli stworzyć dwa oddzielne pliki nagłówkowe. Utrzymywanie dwóch oddzielnych plików nagłówkowych, które zawierają przeważnie identyczne definicje nie jest dobrym pomysłem .Rozwiązania dostarcza dyrektywa externdef.

`Wewnątrz naszych plików nagłówkowych powinniśmy stworzyć segment definicji, które odpowiadają tym zawartym w modułach. Trzeba by włożyć dyrektywę externdef do wewnątrz tego samego segmentu w którym symbol jest w rzeczywistości zdefiniowany. Wiążemy wartość segmentu z symbolem, żeby MASM mógł właściwie zrobić stosowna optymalizację i inne obliczenia w oparciu o pełen adres symbolu:

```
;Plik "HEADER.A":

cseg        segment      para  public  'code'
            externdef    Routine1:near, Routine2:far
cseg        ends
dseg        segment      para  public  'data'
            externdef    i:word, b:byte, flag:byte
dseg        ends
```

Tekst ten adoptuje konwencję z Biblioteki Standardowej UCR stosowania przyrostka „.a" dla plików nagłówkowych asemblera innym popularnymi przyrostkami są „.inc" i „.def"

8.21 PLIKI MAKE

Chociaż zastosowanie oddzielnej kompilacji redukuje czas asemblacji i promuje kod wielokrotnego stosowania i modularność, nie jest bez wad. Przypuśćmy, że mamy program, który składa się z dwóch modułów: pgma.asm i pgmb.asm. Przypuśćmy również, że mamy już zasemblowane oba moduły, więc istnieją pliki pgma.obj i pgmb.obj.W końcu robimy zmianę w pgma.asm i pgmb.asm i assemblujemy pgma.asm ale zapominamy zasemblować plik pgmb.asm. Dlatego też plik pgmb.obj będzie nieaktualny ponieważ ten plik obiektowy nie odzwierciedla zmian dokonanych w pliku pgmb.asm. Jeśli zlinkujemy razem moduły programu, wynikowy plik .exe będzie zawierał zmiany pliku pgma.asm, nie będzie mógł uaktualnić kodu obiektowego powiązanego z pgmb.asm. Jeśli projekt staje się większy ,zawiera więcej modułów z nim związanych, i więcej programistów zaczyna pracę nad projektem, staje się bardzo trudne śledzenie które moduły obiektowe są aktualne.

Ta złożoność normalnie powoduje reasemblację (lub rekompilację) wszystkich modułów w projekcie, nawet jeśli wiele z plików .obj jest aktualnych, po prostu może wydać się zbyt trudne do śledzenia ,które moduły są aktualne a które nie .Robiąc to wyeliminujemy wiele korzyści jakie oferuje oddzielna kompilacja. Na szczęście, jest narzędzie, które może pomóc nam zarządzać dużym projektem – nmake. Program nmake, z małą naszą pomocą, może wyliczyć, które pliki muszą być reasemblowane a które pliki maja aktualne pliki .obj .Z właściwie zdefiniowanym plikiem make, możemy łatwo assemblować tylko te moduły, które absolutnie muszą być zasemblowane dla wygenerowania spójnego programu.

Plik make jest plikiem teekstowym, który wymienia zależności czasu asemblacji pomiędzy plikami. Plik .exe, na przykład, jest zależny od kodu źródłowego którego asemblacja tworzy plik wykonywalny. Jeśli uczynimy zmianę w kodzie źródłowym,(prawdopodobnie) będziemy musieli reasemblować lub rekompilować kod źródłowy tworząc nowy plik .exe.

Typowe zależności obejmują:

- Plik wykonywalny (.exe) generalnie zależy tylko od zbioru plików obiektowych (.obj),które linker łączy do postaci wykonywalnej
- Dany plik kodu wynikowego (.obj) zależy od pliku źródłowego asemblera, który zasemblowany dał plik obiektowy. Obejmuje to plik źródłowy asemblera (.asm) i inne pliki obejmowane podczas tej asemblacji (generalnie pliki .a)
- Pliki źródłowe i pliki include nie zależą od niczego

Plik make ogólnie składa się z instrukcji zależności określonych przez zbiór komend operujących tymi zależnościami. instrukcja zależności przybiera następującą formę:

dependent-file : lista plików

Przykład:

Pgm.exe: pgma.obj pgmb.obj

Ta instrukcja mówi,że"pgm.exe" jest zależny od pgma.obj i pgmb.obj. Każda zmiana która wystąpi w pgma.obj lub pgmb.obj będzie wymagała wygenerowania nowego pliku pgm.exe.

Program nmake.exe stosuje oznaczenia czas /data do określenia czy plik zależny jest nieaktualny w stosunku do pliku od którego zależy. W każdej chwili możemy dokonać zmiany pliku. MS-DOS i Windows uaktualnią zmodyfikowany czas i datę powiązanych z plikiem. program nmake.exe porównuje zmodyfikowane

oznaczenie czas/ data pliku zależnego z modyfikacją oznaczenia czas /data pliku od którego zależy. Jeśli modyfikacja czas/ data pliku zależnego jest wcześniejsza niż jednego lub więcej plików od których zależy, lub jeden plik od których zależy nie jest obecny, wtedy nmake.exe zakłada, że jakaś operacja musi być konieczna dla uaktualnienia pliku zależnego

Kiedy uaktualnienie jest konieczne,nmake.exe wykonuje zbiór poleceń (MS-DOS) instrukcji zależności .Przypuszczalnie te polecenia będą robiły co konieczne do wytworzenia pliku uaktualnionego.

Instrukcje zależności muszą zaczynać się w kolumnie jeden. Każde polecenie, które musi my wykonać dla rozwiązania zależności musi zaczynać się od linii bezpośrednio następującej po instrukcji zależności a każde polecenie musi zaczynać się od akapitu jednym tabem. Instrukcja Pgm.exe wyglądałaby prawdopodobnie następująco:

```
Pgm.exe:pgma.obj pgmb.obj
	ml /Fepgm.exe pgma.obj pgmb.obj
```

(opcaj /Fepgm.exe podaje MASMowi nazwę pliku wykonywalnego „pgm.exe")

Jeśli musimy wykonać więcej niż jedno polecenie rozwiązujące zależności, możemy umieścić kilka poleceń po instrukcji zależności we właściwym porządku. Zauważmy, że musimy zacząć jednym tabem od akapitu wszystkie polecenia.Nmake.exe ignoruje każdą pustą linię w pliku make .Dlatego też, możemy dodać puste linie czyniąc plik łatwiejszym do czytania i zrozumienia

Może być więcej niż jedna instrukcja zależności w pliku make. W powyższym przykładzie na przykład,pgm.exe zależy od plików pgma.obj i pgmb.obj. Oczywiście pliki .obj zależą od plików źródłowych które je generują. Dlatego też przed próbą rozwiązania zależności dla pgm.exe,nmake.exe najpierw zweryfikuje resztę plików make aby zobaczyć czy pgma.obj i pgmb.obj zależą od jakiegoś. Jeśli tak,nmake.exe rozwiąże te zależności najpierw. Rozpatrzmy poniższy plik make:

```
pgm.exe: pgma.obj pgmb,obj
	ml /Fepgm.exe pgma.obj pgmb.obj
pgma.obj:pgma.asm
	ml /c pgma.asm
pgmb.obj"pgmb.asm
	ml /c pgmb.asm
```

Program nmake.exe najpierw przetworzy zależności liniowe, które znajduje w pliku. Jednak, pliki pgm.exe zależą od nich samych maja zależności liniowe. Dlatego też,nmake.exe najpierw zapewnia, że pgma.obj i pgmb.obj są aktualne, przed próba wykonania MASM zlinkuje te pliki razem. Dlatego ,jeśli jedyną zmianę uczyniono w pgmb.asm,nmake.exe podejmie następujące kroki (zakładając, że pgma.obj istnieje i jest aktualny).

1. Nmake.exe przetwarza najpierw instrukcje zależności. Zauważa, że linie zależności istnieją dla pgm.exe (pliki od których zależy pgm.exe)Więc przetwarza najpierw te instrukcje.
2. Nmake.exe przetwarza linię zależności pgma.obj. Zauważa że plik pgma.obj jest nowszy niż plik pgma.asm, więc nie wykonuje następnych poleceń tej instrukcji zależności
3. Nmake przetwarza linię zależności pgmb.obj. Zauważa ,że pgmb.obj jest starszy niż pgma.asm (ponieważ właśnie zmieniliśmy plik źródłowy pgmb.asm)Dlatego też,nmake.exe wykonuje następne polecenie DOS w następnej linii. Generuje to nowy plik pgmb.obj, który jest teraz aktualny.
4. Mając przetworzone zależności pgma.obj i pgmb.obj,nmake.exe teraz zwraca swoją uwagę do pierwszej linii zależności. Ponieważ nmake.exe stworzył nowy plik pgmb.obj, jego oznaczenie czas/ data będzie nowsze niż pgm.exe. Dlatego nmake.exe wykona polecenie ml, które linkuje pgma.obj i pgmb.obj razem do postaci nowego pliku pgm.exe

Zauważmy ,że właściwie napisany plik make poinformuje nmake.exe do asemblacji tylko tych modułów absolutnie koniecznych do wytworzenia spójnego pliku wykonywalnego. W powyższym przykładzie,nmake.exe nie przeszkadza zasemblować pgma.asm ponieważ jego plik wynikowy już był aktualny.

Jest jedna końcowa rzecz godna podkreślenia w związku z zależnościami. Często, pliki wynikowe są zależne nie tylko od pliku źródłowego ,który tworzy pliki wynikowe ,ale każdego pliku, który również zawiera plik źródłowy. W poprzednim przykładzie (najwyraźniej) nie było takich plików include .Nie ma często takich przypadków. Bardziej typowy plik make może wyglądać jak poniższy:

```
pgm.exe: pgma.obj pgmb.obj
	ml /Fepgm.exe pgma.obj pgmb.obj
pgma.obj: pgma.asm pgm.a
	ml /c pgma.asm
pgmb.obj: pgmb.asm pgm.a
	ml /c pgmb.asm
```

Zauważ, że każda zmiana pliku pgm.a wymusza na nmake.exe reasemblację obu pgma.asm i pgmb.asm ponieważ pliki pgma.obj i pgmb.obj oba zależą od pliku dołączanego pgm.a. Pozostawienie pliku dołączanego z listą zależności jest powszechnym błędem programistów, co może tworzyć wewnętrznie sprzeczne pliki .exe.

Zauważmy ,że zwykle nie musimy wyszczególniać plików dołączanych Biblioteki Standardowej UCR ani plików Biblioteki Standardowej .lib w liście zależności. Prawda, nasz wynikowy plik .exe nie zależy od tego kodu, ale Biblioteka Standardowa rzadko się zmienia więc bezpiecznie możemy ją wyrzucić z naszej listy zależności. Czynimy modyfikację Biblioteki standardowej po prostu usuwając każdy stary plik .exe i .obj i wymuszając reasemblację całego systemu.

Nmake.exe, domyślnie, zakłada, że będzie przetwarzany plik make nazwany „make-file". Kiedy uruchamiamy nmake.exe, szuka on „makefile" w bieżącym folderze. Jeśli go nie znajdzie, kończy .Dlatego, jest dobrym pomysłem kolekcjonowanie plików dla każdego projektu nad którym pracujemy w jego własnym podkatalogu i nadanie każdemu projektowi własny makefile. Potem tworząc plik wykonywalny, potrzebujemy tylko zamienić stosowny podkatalog i uruchomić program nmake.exe.

Chociaż ta sekcja omawia program nmake w wystarczających szczegółach do operowania większością projektów nad którymi pracujemy, pamietajmy, że nmake.exe dostarcza znacznej funkcjonalności ,której ten rozdział nie omawia .Dla nauczenia się więcej o programie nmake zajrzyj do dokumentacji, której dostarcza MASM.

8.25 PODSUMOWANIE

Rozdział ten wprowadził kilka dyrektyw asemblerowych i pseudo-opcodów wspieranych przez MASM .Rozdział ten, w żaden sposób nie jest kompletnym opisem tego co MASM może zaoferować.

Instrukcje języka asemblera są swobodnego formatu i jest zwykle jedna instrukcja na linie w pliku źródłowym Chociaż MASM pozwala na wprowadzenie swobodnego formatu powinniśmy ostrożnie konstruować nasz plik źródłowy czyniąc go łatwiejszym do odczytu.

*Zobacz „Instrukcje Języka Asemblera"

MASM śledzi offset instrukcji lub zmiennej w segmencie używając licznika lokacji. MASM zwiększa licznik lokacji o jeden dla każdego bajtu kodu wynikowego napisanego dla pliku wyjściowego.

*Zobacz „Licznik Lokacji"

Podobnie jak HLL'e ,MASM pozwala nam stosować nazwy symboliczne dla zmiennych i etykiet instrukcji. Działać z symbolami jest dużo prościej niż offsetami liczbowymi w programie asemblerowym. Symbole MASMa wyglądają wszystkie jak w HLL'ach z wyjątkiem kilku rozszerzeń.

*Zobacz „Symbole"

MASM dostarcza kilku różnych typów stałych literowych wliczając w to binarne, dziesiętne i heksadecymalne stałe całkowite ,stałe łańcuchowe i stałe tekstowe

*Zobacz „Stałe Literowe"
*Zobacz "Stałe Całkowite"
*Zobacz „Stałe Łańcuchowe"
*Zobacz „Stałe Tekstowe"

Do pomocy przy manipulowaniu segmentami wewnątrz naszego programu MASM dostarcza dyrektyw segment /ends. Z dyrektywą segment możemy sterować porządkiem ładowania i ustawieniem modułów w pamięci.

*Zobacz „Segmenty"
*Zobacz „Nazwy Segmentów"
*Zobacz „Porządek Ładowania Segmentów"
*Zobacz „Operandy Segmentów"
*Zobacz „Typ CLASS"
*Zobacz „Definicje typowych segmentów"
*Zobacz „Dlaczego chcemy sterować porządkiem ładowania"

MASM dostarcza dyrektyw proc /endp dla deklarowania procedur wewnątrz naszych asemblerowych programów Chociaż ,nie ściśle koniecznie, dyrektywy proc/ endp czynią nasz program dużo łatwiejszym do odczytu i pielęgnacji. Dyrektywy proc /endp również pozwalają nam stosować nazwy i lokalnych instrukcji wewnątrz naszych procedur

*Zobacz „Procedury"

Dyrektywy równości pozwalają nam zdefiniować stałe symboliczne różnego rodzaju w naszym programie .MASM dostarcza trzech dyrektyw dla definiowania takich stałych: „=",equ i textequ. Podobnie jak przy HLL'ach, rozsądne stosowanie dyrektyw równań może pomóc uczynić nasz program dużo łatwiejszym do czytania

*Zobacz „Deklarowanie Jawnych Stałych Używając dyrektyw równań"

Jak widzieliśmy w Rozdziale Czwartym .MASM daje nam zdolność deklarowania zmiennych w segmencie danych stosując byte, word, dword i inne dyrektywy. MASM jest asemblerem ze ścisłą kontrolą typów i

przyłącza również typ jako lokację do nazwy zmiennej (większość asemblerów dołącza tylko lokację) To pomaga MASMowi lokalizować niejasne błędy w naszych programach

*Zobacz „Zmienne”
*Zobacz „Typ Label”
*Zobacz „Jak nadać symbol poszczególnym typom”
*Zobacz „Wartości etykiet”
*Zobacz „konflikty typów”

MASM wspiera wyrażenia adresowe, które pozwalają nam stosować operatory arytmetyczne do budowania stałych wartości adresów w czasie asemblacji. Pozwala to również nam przesłonić typ wartości adresu i wyodrębnić róże kawałki informacji o symbolu.

*Zobacz „Wyrażenia adresowe”
*Zobacz „Typy symboli i tryby adresowania”
*Zobacz „Operatory Arytmetyczne i Logiczne”
*Zobacz „Koercja”
*Zobacz :Operatory typu’
*Zobacz „Operatory Pierwszeństwa”

MASM dostarcza kilka udogodnień dla powiadomienia asemblera który segment powiązać z którym rejestrem segmentowym. Daje to nam również zdolność do uchylenia domyślnego wyboru .Pozwala to naszemu programowi zarządzać kilkoma segmentami od razu z minimalnym zamieszaniem

*zobacz „Przedrostki Segmentu”
*Zobacz „Sterowanie Segmentami dyrektywa Assume”

MASM dostarcza nam z „asemblacją warunkową” zdolność, która pozwala nam wybierać które segmenty kodu są w rzeczywistości asemblowane podczas procesu asemblacji. Jest to użyteczne przy wprowadzaniu kodu debuggującego do naszego programu (który możemy łatwo usunąć pojedynczą instrukcją) i dla pisania programów, które muszą działać w różnych środowiskach (poprzez wprowadzanie i usuwanie różnych sekwencji kodu)

*Zobacz „Asemblacja warunkowa”
*Zobacz „Dyrektywa IF”
*Zobacz „Dyrektywa IFE’
*Zobacz „IFDEF i IFNDEF”
*Zobacz IFB,IFNB”
*Zobacz „IFIDN,IFDIF,IFIDNI i IFDIFI’

MASM silnych udogodnień w postaci makr. Makra są częściami kodu, który możemy replikować poprzez proste umieszczanie nazw makr w naszym kodzie. Makro ,zastosowane właściwie, może pomóc napisać krótszy, łatwiejszy do odczytu i bardziej silniejsze programy. Niestety ,niewłaściwie zastosowane, makra tworzą trudne do pielęgnacji, niewydajne programy.

*Zobacz „Makra”
*Zobacz „Makra proceduralne”
*Zobacz „Dyrektywa LOCAL”
*Zobacz „Dyrektywa EXITM”
*Zobacz „Makra: Dobre i Złe wieści”
*Zobacz „Operacje repeat”

MASM dostarcza kilku dyrektyw, które możemy zastosować do tworzenia „listingu zassemblowanego” lub wydruków z naszego programu z dużą ilością generowanych (użytecznych) informacji asemblera. Dyrektywy te pozwalają nam włączać lub wyłączać operacje listingu ,wyświetlają informacje na wyświetlaczu podczas asemblacji ustawiają tytuły na wydrukach wyjściowych

*Zobacz „Sterowanie listingiem”
*Zobacz „Dyrektywy ECHO i %OUT”
*Zobacz „Dyrektywa TITLE”
*Zobacz „:Dyrektywa SUBTTL”
*Zobacz „Dyrektywa PAGE”
*Zobacz „Dyrektywy .LIST, .NOLIST i .XLIST”
*Zobacz „ Inne Dyrektywy Listingu”

Manipulowanie dużymi projektami („Programming in the Large”) wymaga oddzielnej kompilacji (lub oddzielnej asemblacji w przypadku MASMa). MASM dostarcza kilku dyrektyw które pozwalają nam łączyć pliki źródłowe podczas asemblacji, oddzielnie assemblować moduły i przekazywać procedury i nazwy zmiennych pomiędzy modułami.

*Zobacz „Zarządzanie Dużymi Programami”
*Zobacz „Dyrektywa INCLUDE”
*Zobacz „dyrektywy PUBLIC,EXTERN i EXTRN”

*Zobacz „Dyrektywa EXTERNDEF”

8.26 PYTANIA

1) Jaka jest różnica między następującymi sekwencjami instrukcji?
   MOV AX, VAR+1
   a
   MOV AX, VAR
   INC AX
2) Jaka jest format linii źródłowej dla instrukcji asemblera
3) Jakie jest zadanie dyrektywy ASSUME?
4) Co to jest licznik lokacji?
5) Który z poniższych symboli są poprawne?

a) ThisIsASymbol	b) This_Is_A_Symbol
c) This.Is.A.Symbol	d) .Is_This_A_symbol?
e) -----------------------	f) @_$?_To_You
g) 1WayToGo	h) %Hello
i) F000h	j) ?A_0$1
k) $1234	l) Hello there

6) Jak wyszczególniamy segment w porządku ładowania?
7) Jaki jest typ symboli zadeklarowanych poniżej instrukcji?

a) symbol1	equ	0
b) symbol2:		
c) symbol3	proc	
d) symbol4	db	?
e) symbol5	dw	?
f) symbol6	proc	far
g) symbol7	equ	this word
h) symbol8	equ	byte ptr symbol7
i) symbol9	dd	?
j) symbol10	macro	
k) symbol11	segment	para public ‘data’
l) symbol12	equ	this near
m) symbol13	equ	‘ABCD’
n) symbol14	equ	<MOV AX, 0>

8) Którym z symboli z pytania 7 nie są przypisane wartości z bieżącego licznika lokacji?
9) Wyjaśnij zadania poniższych operatorów;
   a) PTR b) SHORT c)THIS d) HIGH e)LOW f) SEG g) OFFSET
10) Jak jest różnicą między makrem REPEAT a operatorem DUP?
11) Jak jest różnica między wartościami ładowanymi do rejestru BX w poniższej sekwencji kodu?

```
        mov     bx, offset Table
        lea     bx, Table
```

12) W jakiej kolejności będą ładowane poniższe segmenty do pamięci?

```
CSEG        segment   para public  ‘CODE’
            -
CSEG        ends
DSEG        segment  para  public  ‘DATA’
            -
DSEG        ends
ESEG        segment  para public  ‘CODE’
            -
ESEG        ends
```

13) Które z następujących wyrażeń adresowych nie tworzy takiego samego wyniku jak inne:

   a) Var1[3][5] b) 15 [Var1] c) Var1[8] d) Var1+2[6] e) Var1*3*5 f) Var1 +3+5