ROZDZIAŁ PIĄTY: **ZMIENNE I STRUKTURY DANYCH**

Rozdział Pierwszy omawiał podstawowe formaty danych w pamięci. Rozdział Trzeci omawiał jak system komputerowy fizycznie organizuje te dane. Ten rozdział kończy to omawianie poprzez połączenie koncepcji reprezentacji danych z ich rzeczywistą fizyczną reprezentacją. Jak sugeruje tytuł, ten rozdział, zajmie się dwoma głównymi tematami: zmiennymi i strukturami danych. Ten rozdział nie zakłada, że mamy jakąś znajomość struktur danych ,chociaż taka umiejętność byłaby użyteczna.

5.0 WSTĘP

Ten rozdział omawia jak deklarować i uzyskać dostęp do zmiennych skalarnych, całkowitych, rzeczywistych, typów danych, wskaźników, tablic i struktur. Musimy opanować te tematy przed przejściem do następnego rozdziału. Deklarowanie i dostęp do tablic, wydaje się być problematyczne dla początkującego programisty asemblerowego. Jednakże, reszta tego tekstu zależy od zrozumienia tych struktur danych i ich reprezentacji w pamięci .Nie próbuj się prześlizgiwać przez ten materiał oczekując ,że nauczysz się go jeśli będziesz go potrzebował później .Potrzebujesz go teraz a próbowanie nauczyć się tego materiału wraz z późniejszymi materiałami, tylko pogmatwa ci w głowie.

5.1 KILKA DODATKOWYCH INSTRUKCJI:LEA,LES,ADD i MUL

Celem tego rozdziału nie jest przedstawienie zbioru instrukcji 80x86.Jednak,są cztery instrukcje dodatkowe które okażą swoją przydatność przy omawianiu reszty tego rozdziału. Są to instrukcje **ładuj adres efektywny (lea), ładuj adres dalekiego wskaźnika używając ES (LES), dodawanie całkowite (ADD)** i **mnożenie bez znaku (MUL)**.Te instrukcje razem z instrukcją mov dostarczają wszystkich niezbędnych możliwości przy dostępie do różnych typów danych omawianych w tym rozdziale.

Instrukcja lea przyjmuje formę:

lea reg_{16}, pamięć

reg_{16} jest 16 bitowym rejestrem ogólnego przeznaczenia. Pamięć, jest to komórka pamięci reprezentowana przez bajt mod/reg/rm (poza tym musi być komórką pamięci ,nie może być rejestrem).

Ta instrukcja ładuje do 16 bitowego rejestru offset z komórki wyspecyfikowanej przez operand pamięci. lea ax,1000h[bx][si],na przykład, załaduje ax adresem z komórki pamięci wskazywanej przez 1000h[bx][si].Jest to oczywiście wartość 1000h+bx+si.Lea jest również całkiem użyteczną przy uzyskiwaniu adresu zmiennej. Jeśli mamy gdzieś w pamięci zmienną I, lea bx, I załaduje do rejestru bx adres (offset) z I.

Instrukcja les przyjmuje formę:

les reg_{16}, $pamięć_{32}$

Ta instrukcja ładuje rejestr es i jeden z 16 bitowych rejestrów ogólnego przeznaczenia z wyspecyfikowanego adresu pamięci. Zauważmy, że każdy adres pamięci który możemy wyszczególnić bajtem mod/reg/rm jest prawidłowy, ale tak jak dla instrukcji lea musi to być komórka pamięci nie rejestr.

Instrukcja les ładuje wyszczególniony rejestr ogólnego przeznaczenia słowem spod danego adresu ,ładuje rejestr es z następnego słowa w pamięci. Ta instrukcja, i jej towarzysz lds (który ładuje ds.) są tylko instrukcjami dla maszyn 80386,które manipulują 32 bitami na raz.

Instrukcja add, podobnie jak jej odpowiednik w x86,dodaje dwie wartości w 80x86.Ta instrukcja przyjmuje kilka form. Jest pięć form na których się tu skoncentrujemy. Są to:

```
add     reg, reg
add     reg, pamięć
add     pamięć, reg
add     reg, stała
add     pamięć ,stała
```

Wszystkie te instrukcje dodają drugi operand do pierwszego, sumę zachowując w pierwszym operandzie. Na przykład, add bx,5,oblicza bx:=bx+5.

Ostatnią instrukcją jaką się zajmiemy jest instrukcja mul (mnożenie).Ta instrukcja ma tylko jeden operand i przybiera formę:

```
mul     reg / pamięć.
```

Jest wiele ważnych szczegółów dotyczących mul, które w tym rozdziale pominiemy. Ze względu na omówienie, które nastąpi, założymy, że rejestr lub komórka pamięci jest 16 bitowym rejestrem lub komórką pamięci. W takim przypadku ta instrukcja oblicza dx:ax;=ax*reg/mem. Zauważmy, że nie ma bezpośredniego trybu dla tej instrukcji.

5.2 DEKLAROWANIE ZMIENNYCH W PROGRAMIE JĘZYKA ASSEMBLERA

Chociaż prawdopodobnie się już domyślamy, że komórki pamięci i zmienne są w pewnym stopniu powiązane, ten rozdział nie będzie wychodził poza wyciąganie silnych podobieństw między nimi dwoma. Cóż, czas naprawić tą sytuację. Rozważmy następujący krótki (i bezużyteczny) program pascalowski:

```
program useless (input, output)
var i,j:integer;
begin
        i:=10;
        write (‘Podaj wartość dla j:’);
        readln (j);
        i:=i*j+j*j;
        wrtieln(‘Wynik to’,j);
end.
```

Kiedy komputer wykona wyrażenie i:=10,zrobi kopię wartości 10 i jakoś zapamiętuję tą wartość dla późniejszego użycia. Osiągamy to tak, że kompilator rezerwuje miejsce w pamięci, specjalnie dla wyłącznego użytkowania zmiennej i .Zakładając, że kompilator określił arbitralnie lokację DS:10h dla naszego celu, możemy użyć instrukcji mov ds:[10h],10 aby to osiągnąć. Jeśli i jest szesnastobitowym słowem, kompilator prawdopodobnie przydzieli zmiennej j słowo startowe od lokacji 12h lub 0Eh.Zakładając,że jest to lokacja 12h,drugie wyznaczone wyrażenie w tym programie mogłoby wyglądać jak następuje:

```
mov     ax, ds:[10h]        ;pobiera wartość z I
mul     ds:[12h]            ;mnoży przez j
mov     ds:[10h],ax         ;przechowuje w I (pomija przepełnienie)
mov     ax, ds:[12h]        ;pobiera J
mul     ds:[12h]            ;Oblicza J*J
add     ds:[10h],ax         ;Dodaje I*J+J*J, przechowuje w I
```

Chociaż jest kilka brakujących szczegółów w tym kodzie, jest dosyć prosty i możemy łatwo zobaczyć co będzie robił ten program.

Teraz wyobraźmy sobie 5000 linijek programu takich jak ten, używający zmiennych takich jak: ds:[10h],ds:[12h],ds:[14h] itd. Czy chcielibyśmy umiejscowić wyrażenie, tam gdzie przypadkowo przechowujemy wynik obliczenia w j zamiast i? Dlaczego powinniśmy się martwić nawet ,że zmienna i jest pod lokacją 10h a j pod 12h?Dlaczego nie powinniśmy używać nazw takich jak i i j zamiast niepokoić się o te adresy numeryczne? Wydaje się sensowne przepisanie tego powyższego kodu tak:

```
mov     ax, i
mul     j
mov     i, ax
mov     ax, j
mul     j
add     i, ax
```

Oczywiście możemy tak zrobić w języku asemblera! Istotnie, jedną z podstawowych zalet asemblera

takiego jak MASM, jest to, że pozwala nam na użycie nazw symbolicznych dla komórek pamięci. Ponadto asembler będzie nawet przydzielał lokacje do nazw automatycznie. Nie potrzebujemy się martwić faktem, że zmienna i jest rzeczywiście słowem z komórki pamięci DS:10h chyba ,że jesteśmy ciekawscy.

Nie będzie to dla nas żadna niespodzianka, że ds będzie wskazywał na dseg segment w pliku SHELL.ASM. Istotnie, skonfigurujemy tak ds żeby wskazywał dseg jako jedną z pierwszych rzeczy kiedy wykona się główny program SHELL.ASM. Dlatego też, wszystko co musimy zrobić to powiedzieć asemblerowi żeby zarezerwował jakieś komórki dal naszych zmiennych w dseg i połączył offset wymienionych zmiennych z nazwami tych zmiennych. Jest to bardzo prosty proces i jest tematem kilku następnych sekcji.

5.3 DEKLAROWANIE I DOSTĘP DO ZMIENNYCH SKALARNYCH

Skalarne zmienne przechowują proste wartości. Zmienne i i j z poprzedniej sekcji są przykładem zmiennych skalarnych. Przykładami struktur danych ,które nie są skalarne są tablice ,rekordy ,zbiory i listy .Te ostatnie typy danych są tworzone z wartości skalarnych. Są one typami zbiorowymi .Zobaczymy, typy zbiorowe trochę później ;najpierw musimy nauczyć się czegoś o typach skalarnych.

Aby zadeklarować zmienną w dseg, musimy użyć wyrażenia takiego jak następujące:

ByteVar byte ?

ByteVar jest etykietą. Powinna się zaczynać w pierwszej kolumnie w segmencie dseg (to jest ,między wyrażeniami segmentem dseg i dseg ends).Dowiemy się wszystkiego o etykietach w kilku rozdziałach ,teraz możemy założyć, że większość poprawnych identyfikatorów w Pascalu/C/Adzie jest również ważnymi etykietami języka asemblera.

Jeśli potrzebujemy więcej niż jedną zmienną w naszym programie, wprowadzamy dodatkową linię w segmencie dseg deklarującą te zmienne. MASM automatycznie przydzieli unikalne lokacji dla zmiennych (czyż nie byłoby zbyt dobrze mieć i i j umiejscowione teraz pod tym samym adresem?).Po deklaracji wymienionych zmiennych, MASM pozwala nam odnosić się do tych zmiennych przez nazwy zamiast przez lokacje w programie .Na przykład, po wprowadzeniu powyższych wyrażeń do segmentu danych (dseg),możemy używać instrukcji takich jak mov ByteVar, al. w programie.

Pierwszej zmiennej, którą umieścimy w segmencie danych zostaje przydzielona komórka pamięci DS:0.Następnej zmiennej w pamięci zostaje przydzielona komórka za poprzednią zmienną. Na przykład, jeśli zmienna spod lokacji zero była zmienną bajtową ,następnej zmiennej zostaje przydzielona komórka pamięci spod DS:1.Jednak,jeśli pierwsza zmienna była słowem ,drugiej zmiennej zostaje przydzielona komórka pamięci DS:2 .MASM zawsze uważnie przydziela zmienne ,w taki sposób aby one na siebie wzajemnie nie zachodziły .Rozważmy następującą definicję dseg:

```
dseg        segment     para public 'data'
bytevar     byte        ?               ;bajt rezerwuje bajty
worvar      word        ?               ;word rezerwuje słowa
dwordvar    dword       ?               ;dword rezerwuje podwójne słowo
byte2       byte        ?
word2       word        ?
dseg        ends
```

MASM przydziela pamięć dla bytevar dla lokacji DS:0.Ponieważ bytevar jest długości jednego bajta ,następną dostępną komórką pamięci będzie DS:1.MASM,zatem,przydzieli pamięć dla wordvar od lokacji DS:1.Ponieważ słowo wymaga dwóch bajtów ,następna dostępna komórka pamięci po wordvar to DS:3,dla której MASM przydziela dwordvar. Dwordvar jest długości czterech bajtów, więc MASM przydziela pamięć dla byte 2 zaczynając od DS:7.Podobnie MASM przydziela pamięć dla word2 od lokacji DS:8.gdybyśmy wpisali inną zmienną po word2,MASM przydzieliłby dla niej lokację DS:0A.

Kiedy będziemy się odnosić do jednej z powyższych nazw, MASM automatycznie zastąpi stosowny offset. Na przykład, MASM przetłumaczy instrukcję mov ax, wordvar jako mov ax ,ds:[1].Więc teraz możemy używać nazw symbolicznych i kompletnie pominąć fakt, że te zmienne są w rzeczywistości komórkami pamięci z odpowiednimi offsetami w segmencie danych.

5.3.1 DEKLAROWANIE I UŻYWANIE ZMIENNYCH BYTE

Więc po co są właściwie zmienne? Cóż ,możemy oczywiście przedstawiać różne typy danych, które mają mniej niż 256 różnych wartości w pojedynczym bajcie. Obejmuje to kilka ważnych i często używanych typów danych wliczając w to typ danych znakowych, typ danych boolowskich, większość typów danych wyliczeniowych i mały typ danych całkowitych (ze znakiem i bez znaku), wystarczy tylko wymienić.

Znaki w typowym kompatybilnym z IBM systemie używają ośmiobitowego zestawu znaków ASCII/IBM (zobacz „A Zestaw znaków ASCII/IBM „) 80x86 dostarcza bogatego zbioru instrukcji do manipulowania danymi

znakowymi .Nie jest to żadna niespodzianka ,że większość zmiennych bajtowych przechowuje dane znakowe.

Typ danych boolowskich przedstawia tylko dwie wartości :prawda lub fałsz. Zatem, do przedstawienia wartości boolowskich wykorzystujemy pojedynczy bit.Jednakże,80x86 w rzeczywistości chce pracować z danymi przynajmniej o szerokości ośmiu bitów. W rzeczywistości używa ekstra kod do manipulowania pojedynczym bitem zamiast całym bajtem Dlatego ,powinniśmy używać całego bajtu dla przedstawiania wartości boolowskiej. Większość programistów używa wartości zero dla przedstawienia fałszu i jeden dla przedstawiania prawdy. Znacznik zera 80x86 wykonuje testy zero / nie zero bardzo łatwo .Zauważmy ,że ten wybór zera lub nie-zera jest głównie dla wygody. Możemy używać każdej z dwóch wartości (lub dwóch różnych zbiorów wartości) dla przedstawienia prawdy lub fałszu.

Większość języków wysokiego poziomu, które wspierają typ danych wyliczeniowych przekształca je (dla użytku wewnętrznego) na liczby całkowite bez znaku .Pierwsza pozycja na liście to w zasadzie pozycja zero ,druga pozycja na liście to pozycja jeden, trzecia pozycja do dwa itd.) Na przykład, rozważmy następujący pascalowski typ wyliczeniowy:

Kolory = (czerwony,niebieski,zielony,purpurowy,pomarańczowy,żółt,biały,czarny);

Większość kompilatorów Pascala przydzieli wartość zero do czerwonego, jeden do niebieskiego itd.

Później zobaczymy jak w rzeczywistości tworzy się własne dane wyliczeniowe w asemblerze. Wszystko czego potrzebujemy teraz, to jak przydzielić pamięć dla zmiennych które przechowują wartości wyliczeniowe .Ponieważ jest niemożliwe ,aby było więcej niż 256 pozycji danych wyliczeniowych ,możemy użyć pojedynczej zmiennej bajtowej dla przechowywania wartości. Jeśli ,powiedzmy ,mamy zmienną kolor typu kolory użyjemy instrukcji mov color,2,co oznacza to samo co kolor :=zielony w Pascalu.(Później nauczymy się jak używać bardziej sensownych wyrażeń, takich jak mov kolor, zielony dla przydzielenia koloru zielonego do zmiennej kolor).

Oczywiście, jeśli mamy małą wartość całkowitą bez znaku (0..255) lub małą wartość całkowitą ze znakiem (-128..127) pojedyncza zmienna bajtowa jest najlepszym sposobem w większości przypadków **.**Zauważmy ,że większość programistów traktuje wszystkie typy danych z wyjątkiem liczb całkowitych ze znakiem jako wartości nieoznaczone. To znaczy, znaki ,wartości boolowskie, typy wyliczeniowe i liczby całkowite bez znaku są zawsze wartościami bez znakowymi. W bardzo specjalnym przypadku możemy potraktować znak jako wartość że znakiem, ale większość czasu znaki są wartościami bez znakowymi.

Są trzy główne wyrażenia dla deklaracji zmiennej bajtowej w programie. Oto one:

```
identyfikator       db          ?
identyfikator       byte        ?
identyfikator       sbyte       ?
```

Identyfikator przedstawia nazwę naszej zmiennej bajtowej. ”db” jest starszym terminem, z przed pojawienia się MASM 6.x.Zobaczymy,że tej dyrektywy używano w innych programach (zwłaszcza tych, które nie używają MASM 6.x lub późniejszych) ale Microsoft uznał, że będzie to termin przestarzały; powinniśmy w zamian używać deklaracji byte lub sbyte.

Deklaracja byte deklaruje zmienną bajtową bez znaku. Powinniśmy używać tej deklaracji dla wszystkich zmiennych bajtowych z wyjątkiem małych liczb całkowitych ze znakiem. Dla liczb całkowitych ze znakiem używamy dyrektywy sbyte (bajt ze znakiem).

Kiedy zadeklarujemy jakieś zmienne bajtowe w tych wyrażeniach, możemy odnosić się do tych zmiennych wewnątrz naszego programu poprzez ich nazwy:

```
i           db          ?
j           byte        ?
k           sbyte       ?
            -
            -
            -
            mov         i,0
            mov         j,245
            mov         k,-5
            mov         al. ,i
            mov         j, al.
            itd.
```

Chociaż MASM 6,x wykonuje małą ilość sprawdzeń zgodności typów, nie powinniśmy ulec wrażeniu, że język asemblera jest językiem z silną kontrola typów. Faktycznie MASM 6.x sprawdza tylko wartości które przemieszczasz by sprawdzić ,czy będą się mieścić w lokacji docelowej. Wszystkie następujące instrukcje są poprawne w MASM 6.x:

```
mov     k,255
mov     j,-5
mov     i,-127
```

Ponieważ wszystkie z tych zmiennych są zmiennymi wielkości bajtu i wszystkie stale skojarzone i dopasowane do ośmiu bitów, MASM na szczęście zezwala na każde z tych wyrażeń. Jeszcze jeśli patrzymy na nie, są one logicznie niepoprawne. Co to znaczy przesunięcie -5 do zmiennej bajtowej bez znakowej? Ponieważ wartość bajtu ze znakiem musi być z zakresu od -128..127,co się zdarzy, kiedy przechowamy wartość 255 wewnątrz zmiennej bajtowej ze znakiem? Cóż ,MASM, po prostu skonwertuje te wartości do ich ośmiobitowych odpowiedników (-5 staje się 0FBh,255 staje się) FFh [-1],itd.).

Być może późniejsze wersje MASM wprowadzą silniejsze badanie zgodności typów wartości, które wkładamy do tych zmiennych albo nie. Jednakże ,powinniśmy zawsze pamiętać ,że zawsze będzie możliwe pominięcie tego sprawdzenia. Pozwoli nam to na pisanie poprawnych programów. Asembler nie pomaga nam tak jak Pascal czy Ada. Oczywiście, nawet jeśli asembler odrzuci takie wyrażenie, będzie łatwo obejść zgodność typów. Rozpatrzmy następującą sekwencję:

```
mov             al.,-5
-
;jakaś liczba wyrażeń które nie wpływają na AL.
-
mov             j, al.
```

Niestety nie ma sposobu, żeby asembler mógł nas poinformować, że przechowujemy nieprawidłową wartość w j. Rejestry ,z natury rzeczy, nie są znakowe ani bez znakowe. Dlatego też. asembler pozwala przechować rejestr wewnątrz zmiennej bez względu na wartość jaka może być w rejestrze.

Chociaż asembler nie sprawdza czy oba operandy instrukcji są ze znakiem czy bez znaku, z dużą pewnością sprawdza ich rozmiar. Jeśli rozmiar nie zgadza się asembler zgłosi stosowny komunikat błędu. Następujące przykłady są niepoprawne:

```
mov i, ax               ;nie można przenieść 16 bitów do ośmiu
mov i,300               ;300 przekracza 8 bitów
mov k,-130              ;-130 przekracza osiem bitów
```

Możemy zapytać „jeśli asembler rzeczywiście nie rozróżnia wartości ze znakiem i bez znaku, dlaczego zawracamy sobie nimi głowę? Dlaczego nie używać po prostu db cały czas? ”Cóż, są dwie przyczyny. Po pierwsze uczyni to nasze programy łatwiejsze do odczytania i zrozumienia jeśli jasno określimy (poprzez użycie byte i sbyte) które zmienne są ze znakiem a które bez znaku. Po drugie, kto mówił coś ,że asembler ignoruje czy zmienne są ze znakiem czy bez znaku? Instrukcja mov ignoruje ale są inne instrukcje ,które nie ignorują.

W punkcie końcowym warto wspomnieć o sprawach dotyczących deklarowania zmiennych bajtowych .We wszystkich deklaracjach widzimy, że pole operandu instrukcji zawsze zawiera pytajnik. Pytajnik mówi asemblerowi, że zmienna powinna być pozostawiona niezainicjowaną kiedy DOS ładuje program do pamięci. Możemy wyspecyfikować wartość początkową dla zmiennej, która może być ładowana do pamięci przed rozpoczęciem wykonywania programu., poprzez zastąpienie znaku zapytania naszą wartością początkową. Rozważmy następującą deklarację zmiennej bajtowej:

```
i               db              0
j               byte            255
k               sbyte           -1
```

W tym przykładzie, asembler inicjuje odpowiednio i, j i k zerem,255 i -1,kiedy program ładuje się do pamięci. Ten fakt okaże całą swoją użyteczność nieco później, zwłaszcza kiedy będziemy omawiali tablice .Asembler tylko sprawdzi rozmiar operandu Nie sprawdza ,aby upewnić się, że operand dla dyrektywy byte jest pozytywny lub, że wartość pola operandu sbyte jest z zakresu -128..127.MASM pozwala na wartość z zakresu -128..255 w polu operandu każdego z tych wyrażeń.

W przypadku, gdy odniesiemy wrażenie, że nie istnieje rzeczywisty powód używania byte i sbyte w programie, powinniśmy zauważyć, że MASM czasami ignoruje różnice w tych definicjach .Debugger Microsoft CodeView nie. Jeśli zadeklarujemy zmienną jako wartość ze znakiem, CodeView wyświetli go jako taki (wliczając w to znak minus jeśli to konieczne).Z drugiej strony CodeView zawsze wyświetla zmienne db i byte jako wartości dodatnie.

5.3.2 DEKLAROWANIE I UŻYWANIE ZMIENNEJ WORD

Większość programów 80x86 używa wartości słowa dla trzech rzeczy: 16 bitowych wartości całkowitych ze znakiem,16 bitowych wartości całkowitych bez znaku i offsetów (wskaźników).Z pewnością możemy używać słowa

dla mnóstwa innych rzeczy równie dobrze, ale te trzy przedstawiają typ danych słowa w większości programów .Ponieważ słowo jest największym typem danych jakim mogą się posługiwać procesory 8086,8088,80186,80188 i 80286,odkryjemy,że dla większości programów, słowo stanowi podstawę obliczeń. Oczywiście 80386 i późniejsze CPU pozwalają na 32 bitowe obliczenia ,ale wiele programów nie używa tych 32 bitowych instrukcji ponieważ są one ograniczone do uruchamiania na 80386 lub późniejszych CPU.

Używamy wyrażenia dw, word lub sword do deklaracji zmiennej słowa. Następujący przykład zademonstruje ich użycie:

```
NoSignedWord        dw          ?
Unsignedord         word        ?
SignedWord          sword       ?
Initialized0        word        0
InitializedM1       sword       -1
InitializedBig      word        65535
InitializedOfs      dw          NoSignedWord
```

Większość z tych deklaracji jest drobną modyfikacją deklaracji byte ,które widzieliśmy w ostatniej sekcji. Oczywiście ,możemy zainicjować każdą zmienną słowa wartością z zakresu -32768..65535 (związek zakresu dla stałych 16 bitowych ze znakiem i bez znaku).Ostatnia z powyższych deklaracji ,jest nowa .W tym przypadku, etykieta pojawia się w polu operandu (nazwa zmiennej NoSignedWord).Kiedy pojawia się etykieta w polu operandu, asembler zastąpi offset tej etykiety (wewnątrz segmentu zmiennych).Jeśli były one tylko deklaracjami w dseg i pojawiają się w tym porządku ,ostatnia z powyższych deklaracji zainicjuje InitializedOfs wartością zero ponieważ offset NoSignedWord to zero wewnątrz segmentu danych .Ta forma inicjacji jest całkiem użyteczna dla inicjacji wskaźników. Ale więcej o tym temacie później.

Debugger CodeView rozróżnia zmienne dw / word i zmienne sword .Zawsze wyświetla wartość bez znaku jako dodatnią wartość całkowitą. Z drugiej strony ,będzie wyświetlał zmienne sword jako wartości ze znakiem (ze znakiem minus jeśli wartość będzie ujemna).Debuggowanie wspiera jeden z głównych powodów dla jakiego chcesz używać word lub sword

---

5.3.3 DEKLAROWNIE I UŻYWANIE ZMIENNYCH DWORD

Możemy użyć instrukcji dd ,dword i sdword dla deklaracji czterobajtowych wartości całkowitych ,wskaźników i innych typów zmiennych. Takie zmienne używają wartości z zakresu -2,147,483,648..4,294,967,295 (związek z zakresu czterobajtowych zmiennych ze znakiem lub bez znaku).Użyjemy tych deklaracji podobnie jak deklaracji word:

```
NoSignedDWord       dd          ?
UnsignedDWord       dword       ?
SignedDWord         sword       ?
InitBig             dword       4000000000
InitNegative        sdword      -1
InitPtr             dd          InitBig
```

Ostatni przykład inicjuje podwójne słowo wskaźnikiem spod adresu segment :offset zmiennej InitBig.

Jeszcze raz, warte jest podkreślenia, że asembler nie sprawdza typu tych zmiennych kiedy inicjuje je wartościami.. Jeśli wartość mieści się w 32 bitach, asembler ją zaakceptuje. Jednak sprawdzanie rozmiaru jest ściśle egzekwowane. Ponieważ tylko 32 bitowe instrukcje mov na procesorach wcześniejszych niż 80386 mają les i lds, otrzymamy błąd jeśli spróbujemy uzyskać dostęp do zmiennej dword na wcześniejszych procesorach używając instrukcji mov. Oczywiście ,nawet na 80386 nie możemy przenieść 32 bitowej zmiennej do 16 bitowego rejestru, musimy użyć 32 bitowego rejestru. Później ,nauczymy się manipulować 32 bitowymi zmiennymi ,nawet na 16 bitowych procesorach. Do tego czasu, będziemy udawać, że nie możemy.

Zapamiętajmy, że CodeView rozróżnia pomiędzy dd /dword a sdword .Pozwoli nam to zobaczyć rzeczywistą wartość naszych zmiennych jaką mamy kiedy debuggujemy nasz program .CodeView tylko robi to ,jeśli użyjemy właściwej deklaracji dla naszych zmiennych .Zawsze używamy dword dla wartości bez znaku i dd lub dword (dword jest lepsze) dla wartości bez znaku.

5.3.4 DEKLAROWANIE I UŻYWANIE ZMIENNYCH FWORD,QWORD I TBYTE

MASM 6.x również pozwala nam zadeklarować sześciobajtowe ,ośmiobajtowe i dziesięciobajtowe zmienne używające wyrażeń df / fword, dq / qword i dt. tbyte .Deklaracje używające tych wyrażeń były początkowo planowane dla wartości zmiennoprzecinkowych i BCD. Są lepsze dyrektywy dla zmiennych zmiennoprzecinkowych i nie musimy się martwić innymi typami danych które używają tych dyrektyw. To omówienie występuje tylko dla

zasady.

Wyrażenia df /fword są głównie przydatne przy deklarowaniu 48 bitowych wskaźników w 32 bitowym trybie chronionym w 80386 i późniejszych CPU .Chociaż możemy używać tej dyrektywy do stworzenia przypadkowej sześciobajtowej zmiennej, są lepsze dyrektywy do tego celu .Powinniśmy używać tylko dyrektyw dla 48 bitowych dalekich wskaźników 80386.

Dq /qword pozwala nam zadeklarować quadword (ośmio bajtową) wartość .Pierwotnym celem tej dyrektywy było tworzenie 64 bitowych zmiennych zmiennoprzecinkowych o podwójnej precyzji i 64 bitowej zmiennej całkowitej. Są lepsze dyrektywy dla tworzenia zmiennych zmiennoprzecinkowych. Ponieważ 64 bitowa zmienna całkowita, nie jest zbyt często wykorzystywana w CPU 80x86 (przynajmniej nie dotąd, dopóki Intel nie udostępni członków z rodziny 80x86 z 64 bitowymi rejestrami ogólnego przeznaczenia)

Dyrektywa dt /tbyte alokuje 10 bajtową pamięć ,Są dwa rdzenne typy danych w rodzinie 80z87 (koprocesor matematyczny) które używają dziesięciobajtowych typów danych: wartość 10 bajtowa BCD i wartości zmiennoprzecinkowej o rozszerzonej precyzji (80 bitów).Ten tekst całkowicie pomija typ danych BCD .Jeśli chodzi o typ zmiennoprzecinkowy, są lepsze sposoby do ich tworzenia.

5.3.5 DEKLAROWANIE ZMIENNYCH ZMIENNOPRZECINKOWYCH REAL4,REAL8 I REAL10

Są dyrektywy, które powinniśmy używać, kiedy deklarujemy zmienne zmiennoprzecinkowe. Podobnie jak dd. dq i dt te wyrażenia rezerwują cztery, osiem lub dziesięć bajtów. Pole operandu dla tych wyrażeń może zawierać znak zapytania (jeśli nie chcemy inicjować zmiennej) lub może zwierać wartość inicjującą w postaci zmiennoprzecinkowej. Następujące przykłady demonstrują ich używanie:

```
x       real4       1.5
y       real8       1.0e-25
z       real10      -1.2594e+10
```

Zauważ ,że pole operandu musi zawierać ważną stałą zmiennoprzecinkową używając albo dziesiętnej albo heksadecymalnej notacji .W szczególności nie jest dozwolona stała całkowita .Asembler będzie protestował jeśli użyjemy operandu takiego jak:

```
x       real4       1
```

Prawidłowo będzie zmienić pole operandu na”1.0”

Proszę zauważyć ,że potrzeba specjalnego sprzętu dla wykonania operacji zmiennoprzecinkowych (np. chip 80x87 lub 80x86z wbudowanym koprocesorem matematycznym).Jeśli taki sprzęt jest niedostępny ,musimy pisać oprogramowanie dla wykonywania operacji jak zmiennoprzecinkowe dodawanie, odejmowanie, mnożenie itp. .W szczególności nie możemy używać instrukcji add 80x86 dla dodawania dwóch wartości zmienno przecinkowych .W tym tekście będziemy omawiać arytmetykę zmiennoprzecinkową w późniejszych rozdziałach(zobacz „Arytmetyka Zmiennoprzecinkowa”).Pomimo to ,jest właściwe omówić jak zadeklarować zmienne zmiennoprzecinkowe w rozdziale o strukturze danych.

MASM również pozwala nam użyć dd, dq i dt dla deklaracji zmiennych zmiennoprzecinkowych (ponieważ te dyrektywy rezerwują konieczną cztero, ośmio lub dziesięcio bajtową przestrzeń).Możemy nawet zainicjować takie zmienne zmiennoprzecinkowymi stałymi w polu operandu. Ale są dwie główne wady deklarowania zmiennych w ten sposób. Po pierwsze, jako bajty, słowa i podwójne słowa, debugger CodeView wyświetli tylko nasze zmienne zmiennoprzecinkowe właściwie jeśli użyjemy dyrektyw real4,real8 i real10.Jeśli użyjemy dd, dq lub dt, CodeView wyświetli nasze wartości jako cztero-, ośmio- lub dziesięcio bajtowe liczby całkowite bez znaku. innym, potencjalnym dużym problemem z używaniem dd, dq i dt jest to ,że pozwalają nam inicjować i stałymi całkowitymi i zmiennoprzecinkowymi (pamiętamy, że real4,real8 i real10 nie).Teraz widzimy jaka to dobra cecha ,na pierwszy rzut oka. Jednak całkowita reprezentacja dla wartości jeden nie jest tym samym co reprezentacja zmiennoprzecinkowa dla wartości 1.0.Więc jeśli przypadkiem wprowadzimy wartość „1” w pole operandu ,kiedy rzeczywiście miało być „1.0” asembler na szczęście to strawi i da nam nieprawidłowy wynik. W związku z tym powinniśmy zawsze używać wyrażeń real4,real8 i real10 dla deklaracji zmiennych zmiennoprzecinkowych.

5.4 TWORZENIE WŁASNYCH NAZW TYPÓW Z TYPEDEF

Powiedzmy, że po prostu jesteśmy niezadowoleni z nazw, które Microsoft postanowił używać dla deklaracji bajtu, słowa ,podwójnego słowa, real i innych zmiennych. Powiedzmy ,że lubimy nazewnictwo Pascalowe lub nazewnictwo C. Chcemy używać terminów takich jak integer ,float, double, char ,boolean lub jakiekolwiek inne. Gdyby to był Pascal ,moglibyśmy przedefiniować nazwy w sekcji **type** programu .W C moglibyśmy użyć wyrażenia **„#define”** lub **typedef** do wykonania tego zadania. Cóż, MASM 6.x ma swoje własne wyrażenie typedef które również pozwala nam stworzyć aliasy tych nazw. Następujący przykład demonstruje jak wprowadzić jakieś zgodne pascalowskie nazwy do naszego programu w języku asemblera:

```
integer         typedef         sword
char            typedef         byte
boolean         typedef         byte
float           typedef         real4
colors          typedef         byte
```

Teraz możemy zadeklarować nasze zmienne bardziej sensownymi wyrażeniami jak:

```
i               integer         ?
ch              char            ?
FoundIt         boolean         ?
X               float           ?
HouseColor      colors          ?
```

Jeśli jesteśmy programistami ADY,C lub FORTRANa (lub innych języków)możemy wybrać nazwę typu bardziej wygodną. Oczywiście, nie zmienia to ani na jotę sposobu w jaki 80x86 lub MASM reagują na te zmienne, ale pozwala to nam tworzyć programy które są łatwiejsze do odczytu i zrozumienia ponieważ nazwy typów są bardziej komunikatywne niż faktyczny, odpowiedni typ.

Zauważmy, że CodeView szanuje odpowiednie typy danych. Jeśli zdefiniujemy wartość całkowitą jako typ sword, CodeView wyświetli zmienne typu całkowitego jako wartość z znakiem .Podobnie, jeśli zdefiniujemy float w znaczeniu real4,CodeView wyświetli jeszcze poprawnie zmienną float jako czterobajtową wartość zmienno przecinkową.

5.5 TYP DANYCH WSKAŹNIKOWYCH

Niektórzy ludzie odnoszą się do wskaźników jako typu danych skalarnych, inni odnoszą się jako do zbiorowego typu danych. Ten tekst traktuje je jako typ danych skalarnych, pomimo, że wykazują właściwości obu ,skalarnego i zbiorowego typu danych. (po kompletny opis zbiorowych typów danych, zajrzyj do „Zbiorowe Typy Danych").

Oczywiście, zaczniemy od pytania: „Co to jest wskaźnik?" Prawdopodobnie mieliśmy do czynienia ze wskaźnikami po raz pierwszy w Pascalu,. C lub Adzie i prawdopodobnie doszliśmy do wniosku, że są straszne .Prawie każdy ma złe doświadczenia kiedy pierwszy raz zetknął się ze wskaźnikami w językach wysokiego poziomu .Spoko ,bez strachu! Wskaźniki są w rzeczywistości łatwe do opanowania w asemblerze. Poza tym, większość problemów ze wskaźnikami które mieliśmy, nie leżała po stronie samych wskaźników, ale raczej w listach powiązanych i strukturze drzewa danych, które próbowaliśmy z nimi implementować. Z drugiej strony wskaźniki ,mają mnóstwo zastosowań w języku asemblera ,nie mających nic wspólnego z listami powiązanymi, drzewami i innymi strasznymi strukturami danych .Istotnie proste struktury danych, takie jak tablice i rekordy, często wymagają użycia wskaźników. Więc jeśli mamy jakiś głęboko zakorzeniony strach przed wskaźnikami ,zapomnijmy o wszystkim co o nich wiemy. Nauczymy się, jak wspaniałe mogą być rzeczywiście wskaźniki.

Prawdopodobnie najlepszym punktem startu jest zdefiniowanie wskaźnika. Więc dokładnie czym jest ten wskaźnik? Niestety języki wysokiego poziomu jak Pascal mają tendencję do ukrywania prostoty wskaźników za murem abstrakcji. To dodaje złożoności przestraszonym programistom ,ponieważ oni nie rozumieją o co chodzi.

Teraz jeśli boimy się wskaźników ,cóż, zignorujmy je do czasu, kiedy zaczniemy pracować z tablicami. Rozważmy następującą deklarację tablicy w Pascalu:

```
M:array [0..1023] of integer;
```

Nawet jeśli nie znamy Pascala, koncepcja tu przedstawiona jest bardzo łatwa do zrozumienia. M jest tablicą 1024 liczb całkowitych w niej zawartych, indeksowanych od M[0] do M[1023].Każdy z elementów tablicy może przechowywać wartość całkowitą która jest niezależna od wszystkich innych .Innymi słowy, ta tablica daje nam 1024 różnych zmiennych całkowitych, do których odnosimy się poprzez jej numer (indeks tablicy) zamiast przez nazwę.

Jeśli spotkamy program ,który ma wyrażenie M[0]:=100,prawdopodobnie nie musielibyśmy myśleć co się z tym dzieje .Wartość 100 jest przechowywana w pierwszym elemencie tablicy M. Teraz rozważmy następujące dwa wyrażenia:

```
i:=0;       (*zakładamy, że i to zmienna całkowita*)
M[i]:=100;
```

Powinniśmy się zgodzić bez większego wahania ,że te dwa wyrażenia wykonują dokładnie tą samą operację M[0]:=100;.Istotnie,prawdopodobnie chętnie zgodzimy się, że możemy używać każdego wyrażenia całkowitego z zakresu 0..1023 jako indeksów wewnątrz tej tablicy. Następujące wyrażenie wykonuje to samo zadanie jak nasze pojedyncze zadanie dla indeksu zero:

```
i:=5;               (*zakładamy, że wszystkie zmienne są całkowite*)
```

```
j:=10;
k:=50;
m[i*j-k]:=100;
```

„Okay, więc co to jest wskaźnik?" myślimy prawdopodobnie ."Wszystkie te wyniki z zakresu wartości całkowitych 0..1023 są poprawne. Więc co? ".Okay, a co myślisz o tym?

```
M[1]:=0;
M[M[1]}:=100;
```

Łoł! Teraz kilka chwil na przetrawienie .Jednak ,gdy weźmiemy to sobie po woli, nabierze to sensu, i odkryjemy ,że te dwie instrukcje wykonują tą samą operację jaką wykonywaliśmy wcześniej .Pierwsze wyrażenie przechowuje zero w elemencie tablicy M[1].Drugie wyrażenie pobiera wartość z M[1] ,które jest całkowite, więc możemy go użyć jako indeks wewnątrz M.,i użyć tej wartości (zero) do kontroli gdzie jest przechowana wartość 100.

Jeśli zaakceptujemy powyższe jako sensowne, być może dziwaczne, ale użyteczne pomimo to, wtedy nie będziemy mieli problemów ze wskaźnikami. Ponieważ M[1] jest wskaźnikiem! Cóż ,nie całkiem ,ale jeśli zmienimy M na pamięć i potraktujemy tą tablicę jako całą pamięć, to jest dokładna definicja wskaźnika.

Wskaźnik jest po prostu komórką pamięci której wartość jest adresem (lub indeksem ,jeśli wolimy) jakiejś innej komórki pamięci. Wskaźniki są bardzo łatwe do deklarowania i używania w programach asemblerowych. Nie musimy nawet martwić się o indeksy tablicy lub o coś w tym rodzaju. Faktycznie ,jedyną komplikacją jaką będziemy napotykali jest to, że 80x86 wspiera dwa rodzaje wskaźników: bliskie wskaźniki i dalekie wskaźniki.

Bliski wskaźnik jest to 16 bitowa wartość która dostarcza offset do segmentu. Może to być każdy segment ale generalnie używamy segmentu danych (dseg w SHELL.ASM).Jeśli mamy zmienną słowo p ,która zawiera 1000h,wtedy p „wskazuje" komórkę pamięci 1000h w dseg. Uzyskując dostęp do słowa na które wskazuje p ,możemy użyć następującego kodu:

```
mov     bx ,p           ;ładuje BX wskaźnikiem
mov     ax,[bx]         ;pobiera dane na które wskazuje p
```

Przez załadowanie wartości z p do bx, kod ten ładuje wartość 1000h do bx (zakładając, że p zawiera 1000h,a a zatem wskazuje komórkę pamięci 1000h w dseg)Druga z powyższych instrukcji ładuje do rejestru ax słowo zaczynające się w komórce której offset pojawia się w bx. Ponieważ bx zawiera 1000h,więc ax będzie ładowany z komórek DS:1000 i DS:1001.

Dlaczego więc nie ładujemy ax bezpośrednio z komórki 1000h używając instrukcji takiej jak mov ax ,ds:[1000h]? No cóż ,jest mnóstwo powodów. Ale podstawowym powodem jest to, że pojedyncza instrukcja zawsze ładuje ax z lokacji 1000h.O ile nie chcemy się bawić z samomodyfikującym się kodem **,**,nie możemy zmienić komórki z której jest ładowany ax. Poprzednie dwie instrukcje ,jednak, zawsze ładują ax z komórki na którą wskazuje p. Jest bardzo łatwo zmienić to pod kontrolą programu., bez używania kodu samomodyfikującego **.** Faktycznie, prosta instrukcja mov p,2000h sprawi, że te dwie powyższe instrukcje ładują ax z komórki pamięci DS:2000 w następnym czasie w którym się wykonają .Rozważmy następujące instrukcje:

```
lea     bx,i
mov     p,bx
-
-
<Jakiś kod, który opuszczamy>

lea     bx,j
mov     p,bx
-
-
mov     bx,p
mov     ax,[bx]
```

Ten krótki przykład demonstruje dwie ścieżki wykonania tego programu. Pierwsza ścieżka ładuje zmienną p spod adresu zmiennej i (pamiętajmy, lea ładuje bx offsetem drugiego operandu)Druga ścieżka kodu ładuje p adresem zmiennej j. Obie ścieżki wykonania zbiegają się w ostatnich dwóch instrukcjach mov, które ładują ax i lub j w zależności od tego która ścieżka wykonania była zastosowana. Pod wieloma względami jest to jak parametr procedury w językach wysokiego poziomu np. Pascalu. Wykonanie tej samej instrukcji odwołuje się do różnych zmiennych w zależności od tego czyj adres (i lub j) pojawi się w p.

Szesnastobitowe bliskie wskaźniki są małe, szybkie a 80x86 dostarcza wydajnych odwołań do ich

używania. Niestety, mają one jedną poważną wadę - możemy uzyskać dostęp tylko do 64K danych (jeden segment) kiedy używamy bliskich wskaźników .Dalekie wskaźniki przezwyciężają to ograniczenie kosztem stworzenia 32 bitowej długości. Jednakże, dalekie wskaźniki pozwalają nam na uzyskanie dostępu do każdej części danych gdziekolwiek w przestrzeni pamięci. Z tego powodu i z faktu, że Standardowa Biblioteka UCR używa wyłącznie dalekich wskaźników ten tekst będzie używał dalekich wskaźników większość czasu .Ale zapamiętajmy, że jest to decyzja oparta na próbie utrzymania rzeczy prostszymi. Kod, który używa bliskich wskaźników zamiast dalekich będzie krótszy i szybszy.

Dostęp do danych ,do których odnosimy się przez 32 bitowy wskaźnik ,będzie musiał załadować część offsetową (mniej znaczące słowo) wskaźnika do bx, bp, si lub di a część segmentową do rejestru segmentowego (typowo es).Wtedy możemy uzyskać dostęp do obiektu używając trybu adresowania bezpośredniego. Ponieważ instrukcja les jest dogodna do tej operacji, jest to doskonały wybór dla ładowania es i jednego z powyższych czterech rejestrów wartością wskaźnika. Następujący przykładowy kod przechowuje wartość w al w bajcie wskazywanym przez daleki wskaźnik p:

```
les     bx,p        ;ładuje p do ES:BX
mov     es:[bx],al  ;przechowuje dalej al.
```

Ponieważ bliskie wskaźniki są długości 16 bitów a dalekie wskaźniki są długości 32 bitów ,możemy po prostu użyć dyrektyw dw /word i dd /dword do alokowania pamięci dla naszych wskaźników (wskaźniki są z natury bez znakowe, więc nie możemy używać normalnie sword lub sdword dla deklaracji wskaźników).
Jednakże, jest dużo lepszy sposób dla tego celu poprzez użycie wyrażenia typedef. Rozważmy następujące formy:

```
typename    typedef    near ptr  basetype
typename    typedef    far  ptr  basetype
```

W tych dwóch przykładach typename reprezentuje nazwy nowych typów ,które tworzymy ,podczas gdy basetype jest nazwą tego typu, który chcemy stworzyć dla wskaźnika. Spójrzmy na określone przykłady:

```
nbytptr     typedef     near  ptr  byte
fbytptr     typedef     far  ptr  byte
colorsptr   typedef     far  ptr  colors
wptr        typedef     near  ptr  word
intptr      typedef     near  ptr  integer
intHandle   typedef     near ptr  intptr
```

(te deklaracje zakładają, że zostały zdefiniowane typy colors i integer, wyrażeniem typedef).Wyrażenie typedef z operandem near ptr tworzy 16 bitowy bliski wskaźnik. Z operandem far ptr tworzy 32 bitowy daleki wskaźnik. MASM 6.x ignoruje typy bazowe dostarczone po near ptr lub far ptr .Jednak ,CodeView używa typów bazowych by wyświetlić obiekt wskaźnika w jego poprawnej formie. Zauważmy, że możemy używać każdego typu jako bazowego dla wskaźników. Jak zademonstrował ostatni przykład, możemy nawet definiować wskaźnik do innego wskaźnika (uchwyt).CodeView wyświetlał by poprawnie obiekt zmiennej typu intHandle wskazujący na adres.

Z powyższymi typami ,możemy teraz wygenerować zmienną wskaźnikową jak następuje:

```
bytestr         nbytptr     ?
bytesttr2       fbytptr     ?
CurrentCollor   colorsptr   ?
CurrentItem     wptr        ?
Last Int        intptr      ?
```

Oczywiście możemy zainicjować te wskaźniki w czasie asemblowania ,jeśli wiemy gdzie będą wskazywały kiedy program rozpocznie się po raz pierwszy. Na przykład, możemy zainicjować zmienną bytestr offsetem MyString używającym następującej deklaracji:

```
Bytestr     nbytptr     MyString
```

5.6 ZBIOROWE TYPY DANYCH

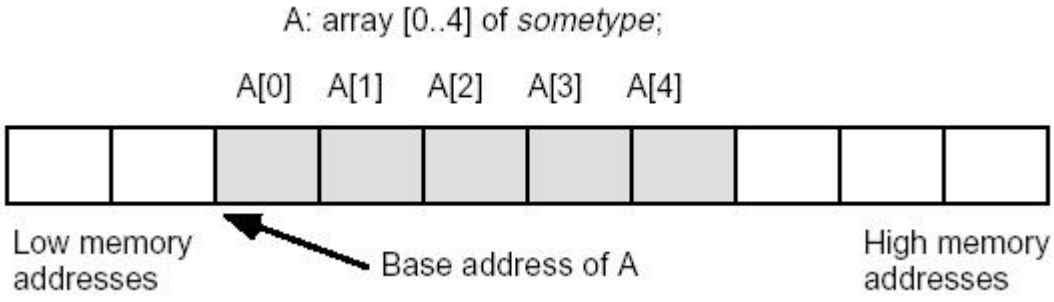
Zbiorowe typy danych są to te zbudowane z innych (głównie skalarnych) typów danych. Tablica jest dobrym przykładem zbiorowego typu danych - jest zbiorem elementów, wszystkich tego samego typu. Zauważmy ,że zbiorowe typy danych nie muszą być złożone ze skalarnych typów danych, są tablice tablic, na przykład, ale ostatecznie możemy rozłożyć zbiorowych typ danych do podstawowego, skalarnego typu.

Ta sekcja omawia dwa z wielu powszechnych zbiorowych typów danych; tablice i rekordy. Jest trochę za wcześnie na omawianie innych bardziej zaawansowanych, złożonych typów danych.

5.6.1 TABLICE

Tablice są prawdopodobnie najbardziej powszechnie używanymi zbiorowymi typami danych. Mimo to większość początkujących programistów ma bardzo słabe pojęcie jak tablice działają i związanymi z nimi sprawami. Zaskakujące jest jak wielu nowicjuszy (a nawet zawodowych) programistów postrzega tablice z kompletnie różnych perspektyw ,kiedy uczą się jak zastosować tablice na poziomie maszynowym.

Abstrakcyjnie ,tablica jest sumą typów danych których członkowie (elementy) są tego samego typu. Wybór elementu z tablicy następuje poprzez indeks całkowity .Różne indeksy wybierają unikalne elementy z tablicy. Ten tekst zakłada, że indeksy całkowite są sąsiadujące



Rysunek 5.1 Implementacja jednowymiarowej tablicy

(chociaż nie jest to wymagane).To znaczy, jeśli numer x jest poprawnym indeksem w tablicy i y jest również poprawnym indeksem, to jeśli x<y, wtedy wszystkie i takie ,że x<i<y są również poprawnymi indeksami w tablicy.

Kiedy stosujemy operator indeksowania do tablicy ,wynikiem jest wyspecyfikowany element tablicy wybrany przez ten indeks .Na przykład ,A[i] wybierze i-ty element z tablicy A .Zauważmy, że nie jest formalnie wymagane ,że element i będzie gdzieś blisko elementu i+1 w pamięci .Pod warunkiem, że A[i] zawsze odnosi się to tej samej komórki pamięci a A [i+1] zawsze odnosi się do odpowiedniej komórki (oby dwie są różne),definicja tablicy jest zadowalająca.

W tym tekście tablice zajmują sąsiadujące komórki w pamięci. Tablica z pięcioma elementami pojawi się w pamięci tak jak pokazano na rysunku 5.1

Adres bazowy tablicy jest adresem pierwszego elementu tablicy i zawsze pojawia się w najmniejszej komórce pamięci. Drugi element tablicy następuje bezpośrednio po pierwszym w pamięci ,trzeci element następuje po drugim itd. .Zauważmy, że nie jest wymagane aby indeksy zaczynały się od zera. Mogą zaczynać się od każdej liczby ,pod warunkiem ,że są one sąsiadujące .Jednak dla celów tego omówienia ,łatwiej będzie omawiać dostęp do elementów tablicy przy pierwszym indeksie równym zero .Ten tekst generalnie rozpoczyna większość tablic od indeksu zero chyba, że jest dobry powód aby było inaczej .Jednak będziemy tak robić konsekwentnie. Nie ma żadnych takich czy innych powodów dla których nie mielibyśmy zaczynać indeksów tablicy od wartości innej niż zero.

Dla uzyskania dostępu do elementów tablicy, potrzebujemy funkcji ,która skonwertuje indeks tablicy na adres elementu indeksowanego. Dla tablicy jednowymiarowej funkcja ta jest bardzo prosta. Oto ona

Adres_Elementu = Adres Bazowy+((Indeks-Indeks_Inicjujący)* Rozmiar_Elementu)

Gdzie Indeks_Inicjujący jest wartością pierwszego indeksu w tablicy (który pomijamy jeśli wynosi zero) a wartość Rozmiar_Elementu jest rozmiarem w bajtach, pojedynczego elementu tablicy.

5.6.1.1 DEKLAROWNIE TABLIC W NASZYM SEGMENCIE DANYCH

Przed uzyskaniem dostępu do elementów tablicy, musimy zarezerwować miejsce w pamięci dla przechowania tej tablicy. Na szczęście deklaracja tablicy wykorzystuje deklaracje jakie widzieliśmy do tej pory .Aby zaalokować n elementów w tablicy, musimy użyć deklaracji tak jak następuje:

arrayname basetype n dup (?)

Arrayname jest nazwą zmiennej tablicowej a basetype jest to typ elementów tej tablicy .To zarezerwuje pamięć do przechowania tablicy .Do uzyskania adresu bazowego tablicy używamy arrayname.

Operand n dup (?) mówi asemblerowi ,żeby powielił obiekt w nawiasach n razy .Ponieważ znak zapytania znajduje się wewnątrz nawiasów, definicja powyżej stworzy n wystąpień nie zainicjowanej zmiennej. Popatrzmy

teraz na kilka określonych przykładów:

```
CharArray       char    128 dup (?)     ;array[0..127] of char
IntArray        integer 8 dup (?)       ,array[0..7] of integer
BytArray        byte    10 dup (?)      ;array[0..9] of byte
PtrArray        dword   4 dup (?)       ;array[0..3] of dword
```

Pierwsze dwa przykłady zakładają, że użyliśmy wyrażenia typedef do zdefiniowania typu danych char i integer.
Te wszystkie definicje alokują pamięć dla nie zainicjowanych zmiennych. Możemy również wyspecyfikować które elementy tablicy będą zainicjowane pojedynczą wartością używając deklaracji podobnej do następującej:

```
RealArray       real4   8 dup (1.0)
IntegerArray    integer 8 dup (1)
```

Obie te definicje tworzą tablice z ośmioma elementami. Pierwsza definicja inicjuje każdą czterobajtową wartość rzeczywistą 1.0,druga deklaracja inicjuje każdy element całkowity jedynką.

Ten mechanizm inicjacji jest spoko jeśli chcemy mieć każdy element tablicy o tej samej wartości. A co jeśli chcemy zainicjować każdy element tablicy różnymi wartościami? Cóż ,jest to również łatwe do wykonania. Deklaracja wyrażeń zmiennych jest taka sama jaką kiedyś widzieliśmy lecz z innym typem formy inicjacji

```
nazwa           type            wartość1,wartość2,wartość3, -,wartośćn
```

Ta forma alokuje n zmiennych typu type .Inicjuje pierwszą pozycję wartością1,drugą pozycję wartością2,itd.Wiec po przez proste wypisanie każdej wartości w polu operandu, możemy stworzyć tablicę z pożądanymi wartościami inicjującymi. W następującej tablicy całkowitej, na przykład, każdy element zawiera kwadrat z jego indeksu:

```
Kwadraty        integer 0,1,4,9,16,25,36,49,64,81,100
```

Jeśli nasza tablica ma więcej elementów niż mieści się w jednej linii, jest kilka sposobów na kontynuowanie tablicy w następnej linii .Najprostszą metodą jest użycie innej instrukcji całkowitej ale bez etykiety:

```
Kwadraty        integer 0,1,4,9,16,25,36,49,64,81,100
                integer 121,144,169,196,225,256,289,324
                integer 361,400
```

Inna opcją, która jest lepsza w danej sytuacji jest użycie lewego ukośnika (backslash) na końcu każdej linii, co powie MASMowi 6.x,żeby kontynuował czytanie od następnej linii.:

```
Kwadraty        integer 0,1,4,9,16,25,36,49,64,81,100, \
                        121,144,169,196,225,256,289,324, \
                        361,400
```

Oczywiście, jeśli nasza tablica zawiera kilka tysięcy elementów ,wypisywanie ich wszystkich nie byłoby raczej zabawne. Większość tablic inicjuje w ten sposób nie więcej niż parę setek danych, ale generalnie, dużo mniej niż 100.

Musimy nauczyć się jednej głównej techniki inicjacji jednowymiarowej tablicy przed pójściem dalej. Rozważmy następującą deklarację:

```
BigArray        word    256 dup (0,1,2,3)
```

Ta tablica ma 1024 elementy ,nie 256.Operand n dup (xxxx) mówi MASMowi aby powielił xxxx n razy ,nie tworzył tablicy z n elementami. Jeśli xxxx składa się z pojedynczej pozycji, wtedy operator dup stworzy tablice n-elementową. Jednak, jeśli xxxx składa się z dwóch pozycji oddzielonych przecinkiem, operator dup stworzy tablicę z 2*n elementami. Jeśli xxxx składa się z trzech pozycji oddzielonych przecinkami, operator dup tworzy tablice z 3*n pozycjami i tak dalej .Ponieważ mamy cztery pozycje w nawiasach powyżej ,operator dup tworzy 256*4 ,lub 1024 pozycji w tablicy. Wartości w tablicy będą inicjowane przez 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3.
Zobaczymy ,wiele możliwości operatora dup kiedy przyjrzymy się wielowymiarowym tablicom trochę później.

5.6.1.2 UZYSKIWANIE DOSTĘPU DO ELEMENTÓW TABLICY JEDNOWYMIAROWEJ

Przy uzyskiwaniu dostępu tablicy zero-based (opartej o zero) możemy użyć uproszczonej formuły:
Adres_Elementu=Adres_Bazowy+Indeks*Rozmiar_Elementu

Dla pola Adres_Bazowy możemy użyć nazwy tablicy (ponieważ MASM kojarzy adres pierwszego operandu z etykietą). Pole Rozmiar_Elementu jest liczbą bajtów każdego elementu tablicy. Jeśli obiektem jest tablica bajtów, pole Rozmiar_Elementu wynosi jeden (prowadzi do tego bardzo proste obliczenie).Jeśli każdy element tablicy jest słowem (lub wartością całkowitą ,lub innym dwu bajtowym typem) wtedy Rozmiar_Elemetu wynosi dwa. I tak dalej. Dla uzyskania dostępu do elementów tablicy Kwadraty z poprzedniej sekcji, użyjemy następującej formuły:

Adres_Elementu+Kwadraty+indeks*2

80x86 koduje równoważnik instrukcji AX:=Kwadraty[indeks] jako

```
mov     bx ,indeks
add     bx, bx          ;sprytny sposób obliczenia 2*bx
mov     ax, Kwadraty[bx]
```

Są dwie ważne rzeczy do zapamiętania. Przede wszystkim, ten kod używa instrukcji add zamiast instrukcji mul do obliczenia 2*indeks.Głównym powodem dla wybrania add jest to, że jest bardziej dogodna (pamiętamy, że mul nie pracuje ze stałymi i działa tylko na rejestrze ax)Zresztą add jest dużo szybsza niż mul na wielu procesorach, ale ponieważ prawdopodobnie nie wiemy tego**,** czy był to wybór właściwy tej instrukcji

Drugą rzeczą do zapamiętania o tej sekwencji instrukcji jest to, że niezbyt wyraźnie oblicza sumę adresu bazowego i indeksu razy dwa. Faktycznie stosuje tryb adresowania indeksowego do pośredniego obliczania tej sumy. Instrukcja mov ax, Kwadraty[bx] ładuje ax z lokacji Kwadraty+bx, która jest adresem bazowym plus indeks * 2 (ponieważ bx zawiera indeks*2).Moglibyśmy użyć

```
lea     ax, Kwadraty
add     bx, ax
mov     ax,[bx]
```

w miejsce ostatniej instrukcji, ale dlaczego używamy trzech instrukcji gdzie jedna zrobi tą samą pracę? Jest to dobry przykład dlaczego powinniśmy znać tryby adresowania „od podszewki" .Wybierając właściwy tryb adresowania możemy zredukować rozmiar naszego programu, tym samym go przyspieszając.

Tryb adresowania indeksowego w 80x86 jest naturalnym dla uzyskiwaniu dostępu do elementów tablicy jednowymiarowej. Istotnie, jego składnia nawet sugeruje dostęp do tablicy **.**Jedną rzeczą do zapamiętania jest to ,że musimy pamiętać o pomnożeniu indeksu przez rozmiar elementu .Nieprawidłowe wykonanie da w efekcie niepoprawny wynik.

Jeśli używamy 80386 lub późniejszych CPU, możemy wykorzystać tryb adresowania indeksowego ze skalowaniem do przyspieszenia uzyskania dostępu do elementów tablicy. Rozważmy następujące instrukcje:

```
mov     ebx, indeks         ;zakładamy 32 bitową wartość
mov     ax, Kwadraty [ebx*2]
```

To zmniejsza wykonywanie programu o 2 instrukcje .Zobaczymy wkrótce ,że dwie instrukcje są niekoniecznie szybsze niż trzy instrukcje ,ale mam nadzieję że rozumiesz o co chodzi **.**Znajomość trybów adresowania może nam z pewnością pomóc.

Przed przejściem do tablic wielowymiarowych, parę dodatkowych punktów o trybach adresowania i tablicach. Powyższa sekwencja pracuje dobrze jeśli chcemy uzyskać dostęp do pojedynczego elementu z tablicy Kwadraty. Jednak, jeśli chcemy uzyskać dostęp do kilku różnych elementów z tablicy w środku krótkiej sekcji kodu i możemy sobie pozwolić „na utratę" innego rejestru dla tej operacji, możemy skrócić nasz kod i, być może, przyspieszyć go. Instrukcja mov ax, Kwadraty[bx] jest długa na cztery bajty (zakładając ,że potrzebujemy dwu bajtowego przemieszczenia do przechowania offsetu Kwadratów w segmencie danych)Możemy zredukować to do instrukcji dwu bajtowej poprzez użycie trybu adresowania bazowego indeksowanego jak następuje:

```
lea     bx, Kwadraty
mov     si, indeks
add     si, si
mov     ax, [bx][si]
```

Teraz bx zawiera adres bazowy a si zawiera wartość indeks*2.Oczywiście,zastąpiło to pojedynczą cztero bajtową instrukcję trzy bajtową i dwu bajtową instrukcją, bardzo dobra zamiana. Jednakże ,nie musimy załadowywać bx adresem bazowym Kwadratów dla następnego dostępu .Następująca sekwencja jest o jeden bajt krótsza niż porównywalna sekwencja która nie ładuje adresu bazowego do bx:

```
lea     bx ,Kwadraty
mov     si indeks
add     si ,si
mov     ax,[bx] [si]
-
-                       ;Założenie :Bx jest zostawione w spokoju
-                       ;bezpośrednio w tym kodzie
mov     si,indeks2
add     si, si
mov     cx, [bx][si]
```

Oczywiście lepszy dostęp do Kwadratów uzyskamy bez załadowywania do bx ,będziemy mieli większe oszczędności .Lekko skomplikowana sekwencja kodu takiego jak ten czasami opłaci się pokaźnie .Jednak

oszczędności zależą wyłącznie od tego jakiego procesora używamy. Sekwencja kodu, która działa szybko na 8086 może rzeczywiście pracować wolniej na 80486 (i vice versa).Niestety, jeśli chodzi nam o szybkość to nie ma żadnych twardych, szybkich reguł **.**Faktycznie jest bardzo trudno przewidzieć szybkość większości instrukcji na prostym 8086,a nawet na procesorach takich jak 80486 i Pentium/80586 które oferują przetwarzanie potokowe, zintegrowaną pamięć podręczną a nawet operacje superskalarne.

5.6.2 TABLICE WIELOWYMIAROWE

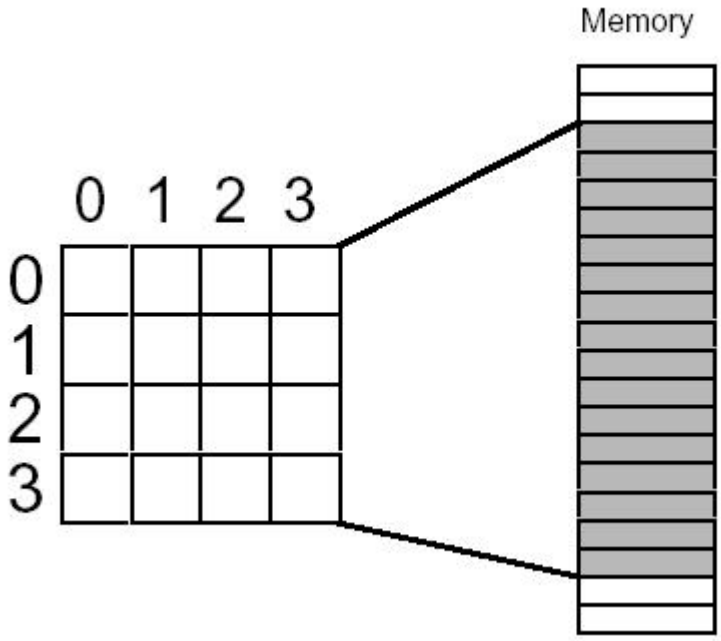
Sprzęt 80x86 może łatwo obsługiwać jednowymiarowe tablice. Niestety, nie ma magicznego trybu adresowania pozwalającego nam łatwo uzyskiwać dostęp do elementów tablic wielowymiarowych. To wymaga sporej pracy i mnóstwa instrukcji.

Przed omówieniem jak zadeklarować lub uzyskać dostęp do tablic wielowymiarowych, będzie dobrym pomysłem wykombinować jak zaimplementować je w pamięci. Pierwszy problem to wymyślić jak przechować wielowymiarowy obiekt w jednowymiarowej przestrzeni pamięci.

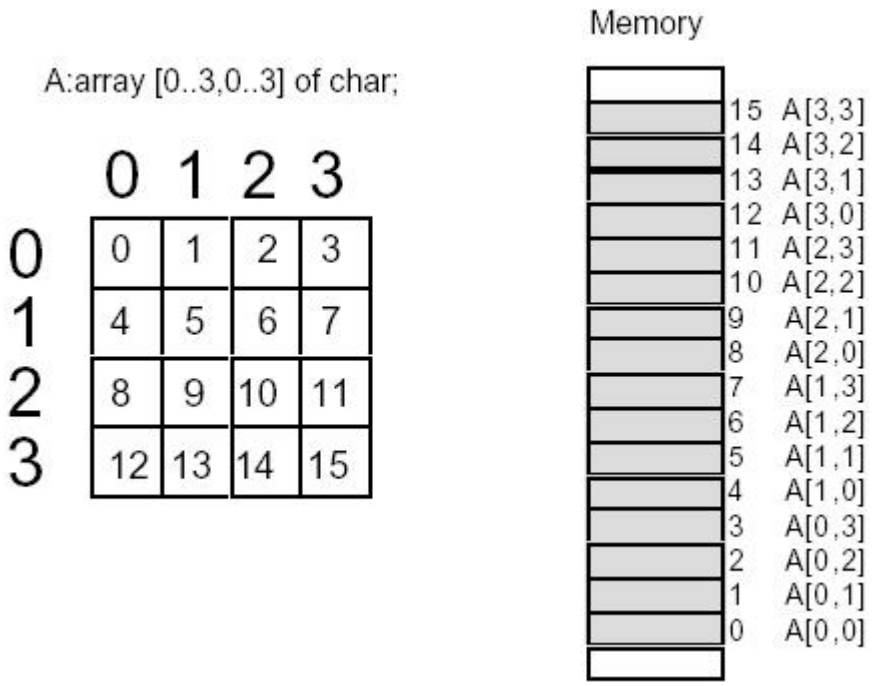
Rozważmy przez chwile tablicę Pascalowską w postaci A:array[0..3,0..3] of char. Ta tablica zawiera 16 bajtów zorganizowanych w 4 wiersze po cztery znaki. Jakoś musimy znaleźć związek z każdym z 16 bajtów w tej tablicy i 16 sąsiadującymi bajtami w pamięci głównej. Rysunek 5.2 pokazuje jeden sposób zrobienia tego.

Rzeczywiste mapowanie nie jest ważne tak długo aż nie wydarzą się dwie rzeczy: (1) każdy element odwzorowuje unikalną komórkę pamięci (to znaczy, żadne dwa wejścia w tablicy nie uzyskują dostępu do tej samej komórki pamięci) i (2) odwzorowywanie jest spójne. To znaczy, dany element w tablicy zawsze odwzorowywuje tą samą komórkę pamięci. Więc to czego potrzebujemy to funkcji z dwoma parametrami wejściowymi (wiersze i kolumny) która wstawi offset do szesnastobitowej liniowej tablicy.

Teraz każda funkcja która wypełni powyższe ograniczenia będzie pracowała dobrze. Istotnie moglibyśmy losowo wybierać odwzorowywanie tak długo jak byłoby unikalne .Jednakże, to co rzeczywiście chcemy odwzorować to wydajne obliczanie czasu wykonania i praca dla każdego rozmiaru tablicy (nie tylko 4x4 lub nawet ograniczenie do dwóch wymiarów).Podczas gdy jest duża liczba możliwych funkcji



Rysunek 5.2: Odwzorowanie tablicy 4x4 w pamięci



Rysunek 5.3: Rzędowe pozycjonowanie elementów

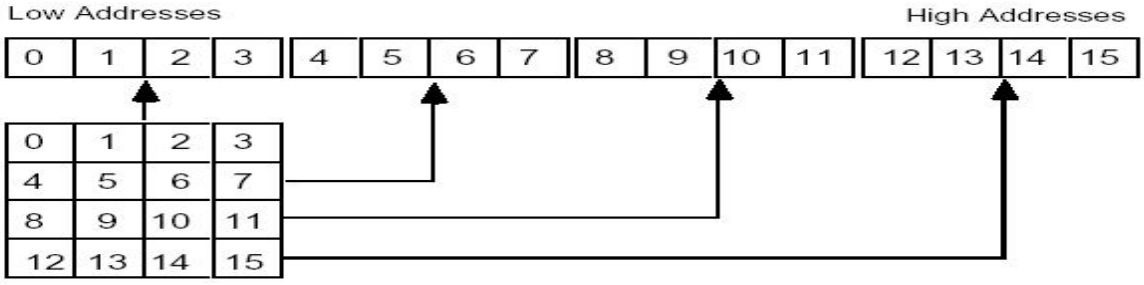
które odpowiednio to wyliczają, są dwie funkcje, których używa większość programistów i języków wysokiego poziomu: Rzędowe pozycjonowanie elementów i kolumnowe pozycjonowanie elementów

---

5.6.2.1Rzędowe pozycjonowanie elementów

Rzędowe pozycjonowanie elementów przydziela następujące po sobie elementy, przesuwając wzdłuż wierszy a potem w dół kolumn, do następujących po sobie komórek pamięci. Odwzorowywanie jest najlepiej pokazane na rysunku 5.3.

Rzędowe pozycjonowanie elementów jest metodą stosowaną przez większość języków wysokiego poziomu, takich jak Pascal,C,Ada,Modula-2 itp. .Łatwo jest zaimplementować i łatwo użyć języka maszynowego (zwłaszcza w debuggerze takim jak CodeView).Konwersja ze struktury dwuwymiarowej do tablicy liniowej jest bardzo intuicyjne.



Rysunek 5.4:Inny widok rzędowego pozycjonowania elementów dla tablicy 4x4

Zaczynamy od pierwszego wiersza (wiersz numer zero) a potem łączymy drugi wiersz do jego końca. Potem łączymy trzeci wiersz do końca listy, potem czwarty wiersz itd. (zobacz rysunek 5.4)

Dla tych którzy lubią myśleć pod kątem kodu programu, następująca zagnieżdżona pętla Pascalowska również demonstruje jak pracuje Rzędowe pozycjonowanie elementów :

```
index:=0;
for colindex := 0 to 3 do
        for rowindex:= 0 to 3 do
        begin
                        memory[index]:=rowmajor[colindex][rowindex];
                        index:=index+1;
        end;
```

Ważną rzeczą do zapamiętania z tego kodu jest to, ,że indeks najbardziej na prawo wzrasta najszybciej .Jest tak ,ponieważ alokujemy kolejne komórki pamięci, następuje przyrost indeksu najbardziej na prawo aż do momentu kiedy dotrzemy do końca bieżącego wiersza. Po dotarciu do końca, przestawiamy indeks z powrotem na początek wiersza i zwiększamy następny sąsiadujący indeks o jeden (to znaczy przechodzimy w dół do następnego wiersza).Działa to równie dobrze dla każdej liczby wymiarów .Następujący Pascalowski segment demonstruje rzędowe pozycjonowanie elementów dla tablicy 4x4x4:

```
Index:=0;
for  depthindex ;=0 to 3 do
        for colindex:=0 to 3 do
                for rowindex:=0 to 3 do begin
                        memory[index]:=rowmajor [depthindex][colindex][rowindex];
                        index:=index+1;
                end;
```

Rzeczywista funkcja która konwertuje listę wartości indeksów do offsetu nie wymaga pętli lub dużych wyliczeń .Istotnie, jest drobna modyfikacja formuły dla obliczenia adresu elementu jednoelementowej tablicy. Formuła oblicza offset dla dwu wymiarowej tablicy rzędowego pozycjonowanie elementów zadeklarowanej jako A:array[0..3,0..3] of integer.

Adres_Elementu=Adres_Bazowy+(colindex*row_size+rowindex)*Rozmiar_Elementu

Jak zwykle ,Adres_Bazowy jest to adres pierwszego elementu tablicy (A[0][0] w tym przypadku) a Rozmiar_Elementu jest rozmiarem pojedynczego elementu tablicy, w bajtach. Colindex jest indeksem lewym ,rowindex jest indeksem prawym w tablicy .Row_size jest liczbą elementów w jednym wierszu (cztery w tym przypadku, ponieważ każdy wiersz ma cztery elementy).Zakładając ,Rozmiar_Elementu jeden, ta formuła oblicza następujący offset z adresu bazowego:

Column Index	Row Index	Offset into Array
0	0	0
0	1	1
0	2	2
0	3	3
1	0	4
1	1	5
1	2	6
1	3	7
2	0	8
2	1	9
2	2	10
2	3	11
3	0	12
3	1	13
3	2	14
3	3	15

Dla trzy wymiarowej tablicy, formuła obliczania offsetu w pamięci jest następująca:

Adres=Baza+((depthindex*col_size+colindex)*row_size+row_index)*Rozmiar_Elementu

Col_size jest to liczba pozycji w kolumnie, row_size jest liczbą pozycji w wierszu. W Pascalu ,jeśli zadeklarujemy tablicę jako „A:array[i..j][k..l[[m..n] of type, formuła obliczania adresów elementów tablicy to:
Adres=Baza+((LeftIndex*depth_size+depthindex)*col_size+colindex)*row_size+rowindex)*Rozmiar_Elementu.
Depth_size równa się i-j+1,col_size i row_size są takie same jak wcześniej. Lewy indeks przedstawia wartość indeksu lewego.

Teraz zobaczymy wzór. Jest to ogólna formuła, która obliczy offset w pamięci dla tablicy o różnych wymiarach,jednak rzadko będziemy używać więcej niż cztery wymiary.

Innym dogodnym sposobem myślenia o tablicach rzędowych jest tablica tablic. Rozważmy następującą definicję jedno wymiarową tablicy:

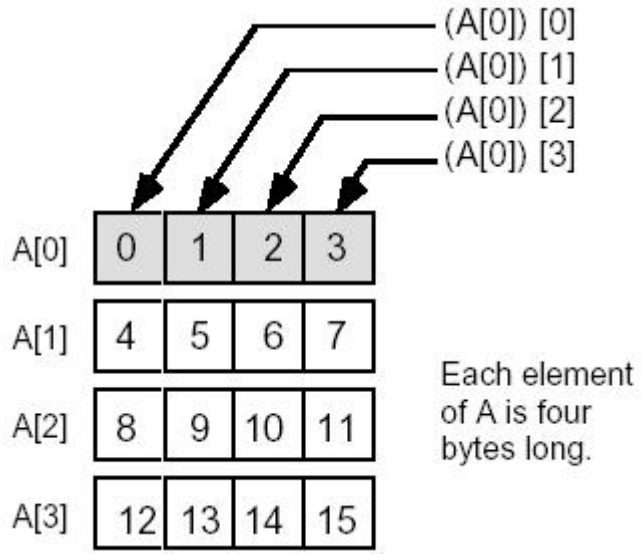
A:array [0..3] of sometype;

Zakładamy, że sometype jest typem „sometype=array[0..3] of char”.

A jest jednowymiarową tablicą.Jej pojedyncze elementy to tablice, ale możemy śmiało zignorować je na razie. Formuła do obliczania adresu elementów jednowymiarowej tablicy to:
Adres_Elementu=Baza+Index*Rozmiar_Elelmentu

W tym przypadku Rozmiar_Elementu wydaje się być cztery ponieważ każdy element z A jest tablicą czterech znaków .Więc co ta formuła oblicza? Oblicza adres bazowy każdego wiersza w tej tablicy znaków 4x4.(zobacz rysunek 5.5).

Oczywiście, raz obliczywszy adres bazowy wiersza, możemy ponownie obliczyć jednowymiarową formułę aby otrzymać adres poszczególnego elementu. Podczas gdy nie wpływa to na obliczenia znacznie, jest prawdopodobnie trochę łatwiej zająć się kilkoma jednowymiarowymi obliczeniami zamiast obliczeniami adresów elementów złożonej wielowymiarowej tablicy.



Rysunek 5.5: Obraz tablicy 4x4 jako Tablica tablic

Rozważmy pascalowską tablicę zdefiniowana jako:

```
type
        OneD = array [0..3] of char;
        TwoD = arra [0..3] of OneD;
        ThreeD = array [0..3] of TwoD;
        FourD - array [0..3] of ThreeD;
var
        A:array [0..3] of FourD;
```

OneD jest rozmiaru czterech bajtów. Ponieważ TwoD zawiera cztery tablice OneD, jej rozmiar to 16 bajtów. Podobnie, ThreeD to cztery TwoD, więc jest długa na 64 bajty. W końcu, FourD to cztery ThreeD, więc jest długa na 256 bajtów. Do obliczania adresu „A[b][c][d][e][f]” możemy użyć sekwencji następujących kroków:

*Obliczamy adres A[b] jako „Baza+b*rozmiar”.Gdzie rozmiar to 256 bajtów. Użyjemy tego wyniku jako nowego adresu bazowego w następnym obliczeniu.

*Obliczamy adres A[b][c] z formuły „Baza+c*rozmiar”, gdzie Baza jest wartością uzyskaną

bezpośrednio powyżej a rozmiar to 64.Użyjemy tego wyniku jako nowej bazy w następnym obliczeniu.
*Obliczamy adres A[b][c][d] przez „Baza+d*rozmiar” z Bazą pochodzącą z obliczenia powyżej a rozmiar wynosi 16.
*Obliczamy adres A[b][c][d][e] z formuły „Baza+e*rozmiar” z Bazą z powyższego obliczenia i rozmiarem cztery. Używamy tej wartości jako bazy dla następnego obliczenia.
*W końcu obliczamy adres A[b][c][d][e][f] używając formuły „Baza+f*rozmiar” gdzie baza pochodzi z powyższego obliczenia a rozmiar to jeden (oczywiście możemy po prostu zignorować końcowe mnożenie)Wynik jaki otrzymaliśmy w tym miejscu jest adresem pożądanego elementu

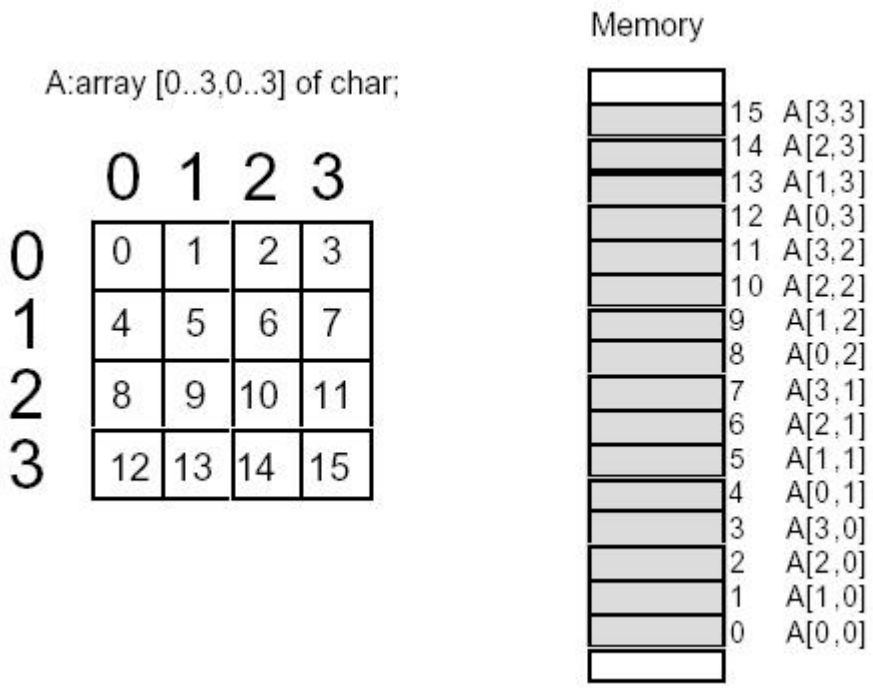
Nie tylko ten schemat jest łatwiejszy do wykonania niż powyższe formuły, lecz także jest łatwiej obliczyć (używając pojedynczej pętli) równie dobrze. Przypuśćmy, że mamy dwie tablice zainicjowane jak następuje

A1={256,64,16,4,1} i A2={b,c,d,e,f}

Wtedy kod pascalowski wykona obliczenia adresów elementów :

```
for i:=0 to 4 do
    base:=base+A1[i]*A2[i];
```

Przypuszczalnie baza zawiera adres bazowy tablicy przed wykonaniem tej pętli. Zauważmy, że możemy łatwo rozszerzyć ten kod do każdej liczby wymiarów poprzez proste ,właściwe, inicjowanie A1 i A2 i zmieniającą się wartość końcową pętli.



Rysunek 5.6 Kolumnowa organizacja elementów

Okazuje się ,że kosztowne obliczenia dla pętli takiej jak ta jest zbyt wspaniałe do rozwiązania w praktyce. Moglibyśmy użyć tylko algorytmu takiego jak ten, jeśli potrzebowalibyśmy wyspecyfikować liczbę wymiarów podczas wykonywania programu.. Istotnie, jednym z głównych problemów, których nie znajdziemy w wysoko wymiarowych tablicach w asemblerze jest to, że asembler wyświetla nie prawidłowości związane z tym adresem .Łatwo jest wprowadzić coś takiego jak „A [b,c,d,e,f] do programu pascalowskiego, nie zdając sobie sprawy co kompilator robi z kodem. Programiści asemblerowi nie są tak nonszalanccy - widzą bałagan który się wprowadza kiedy używają wysoko wymiarowych tablic. Dobry asemblerowy programista próbuje unikać dwu wymiarowych tablic i często ucieka się do sztuczek żeby uzyskać dostęp do danych w takiej tablicy kiedy jej użycie staje się absolutną koniecznością. Ale więcej o tym trochę później.

5.6.2.2 Kolumnowa organizacja elementów

Kolumnowa organizacja elementów jest inną funkcją często używaną do obliczania adresu elementu tablicy.

FORTRAN i różne dialekty BASICa (np. Microsoft) używają tej metody do indeksowania tablic.

W rzędowej organizacji elementów indeks najbardziej na prawo wzrasta szybciej ponieważ przenosimy bezpośrednio kolejne komórki pamięci. W kolumnowej organizacji elementów indeks najbardziej na lewo wzrasta szybciej. Obrazowo, kolumnowa organizacja elementów tablicy jest zorganizowana tak jak pokazano na rysunku 5.6

Formuła do obliczania adresu elementu tablicy kiedy używamy kolumnowej organizacji elementów jest bardzo podobna do tej dla rzędowej organizacji elementów. Po prostu odwracamy indeksy i rozmiary w obliczeniach:

Dla dwu wymiarowej kolumnowej tablicy:

Adres_Elemnetu=Adres_Bazowy+(rowindex*col_size+colindex)*Rozmiar_Elementu

Dla trzy wymiarowej kolumnowej tablicy:

Adres=Baza+((rowindex*col_size+colindex)*depth_size_depthindex)*Rozmiar_Elementu

Dla cztero wymiarowej kolumnowej tablicy:

Adres= Baza+(((rowindex*col_size+colindex)*deoth_size+depthindex)*Left_size+Leftindex)*Rozmiar_Elementu

Pojedyncza pętla pascalowska utrzymuje dostęp do rzędowej tablicy pozostający niezmienny (dostęp do A [b][c][d][e][f]):

```
for i := 0 to 4 do
        base:=base+A1[i]*A2[i];
```

Podobnie, wartość inicjująca tablicy A1 pozostaje niezmienna:

A1={256,64,16,4,1}

Jedynie rzeczą którą musimy zmienić jest wartość inicjująca tablicy A2, a wszystko co musimy tu zrobić to odwrócić porządek indeksów:

A2={f,e,d,c,b}

5.6.2.3 PRZYDZIELANIE PAMIĘCI DLA TABLIC WIELOWYMIAROWYCH

Jeśli mamy tablicę m. x n, będziemy mieli m.*n elementów i wymagane m.*n*Romiar_Elemntu bajtów pamięci. Przydzielając pamięć dla tablicy musimy zarezerwować taką ilość pamięci .Jak zwykle jest kilka różnych sposobów zrealizowania tego zadania. Ten tekst spróbuje wykonać to tak aby było łatwo odczytać i zrozumieć nasze programy.

Zastanówmy się ponownie nad operatorem dup dla rezerwowania pamięci. n dup (xxxx) powtarza xxxx n razy .Jak widzieliśmy wcześniej, ten operator dup pozwala nie tylko jedną, ale kilka pozycji w nawiasach powielić określoną ilość razy. Faktycznie, operator dup pozwala na pracę z tym co możemy spodziewać się znaleźć w polu operandu wyrażenia bajtowego wliczając w to dodatkowe wystąpienia operatora DUP. Rozważmy następującą instrukcję:

```
A       byte    4 dup (4 dup(?))
```

Pierwszy operator dup powtarza wszystko w środku nawiasów cztery razy. Wewnątrz nawiasów operacja 4 DUP (?) mówi MASMowi aby zarezerwował miejsce w pamięci dla czterech bajtów. Cztery kopie czterech bajtów dają 16 bajtów, liczbę konieczną dla tablicy 4x4.Oczywiście,rezerwowanie pamięci dla tej tablicy może być łatwiejsze przez użycie instrukcji

```
A       byte    16 dup (?)
```

Inny sposobem w asemblerze jest zarezerwowanie 16 sąsiednich bajtów w pamięci. Jeśli chodzi o 80x86,nie ma różnic między tymi dwoma formami. Z drugiej strony, dawne wersje dostarczały lepszego znaku, że A to tablica 4x4 niż późniejsze wersje. Późniejsze wersje wyglądają jak jednowymiarowa tablica z szesnastoma elementami.

Możemy bardzo łatwo rozszerzyć to pojęcie tablic z większą liczbę argumentów operatora. Deklaracja dla trójwymiarowej tablicy, A;array[0..2,0..3,0..4] of integer mogłaby być

```
A       integer 3 dup (4 dup (5 dup (?)))
```

(oczywiście, potrzebujemy instrukcji integer typedef word w naszym programie dla tego wykonania)

Ponieważ był już przypadek jednowymiarowych tablic, możemy zainicjować każdy element tablicy specyficzną wartością poprzez zastąpienie znaku zapytania jakąś konkretną wartością. Na przykład, inicjacja powyższej tablicy, żeby każdy element zawierał jedynkę, użyjemy kodu

```
A       integer 3 dup (4 dup (5 dup (1))))
```

Jeśli chcemy zainicjować każdy element tablicy różnymi wartościami ,musimy wprowadzić każdą wartość indywidualnie. Jeśli rozmiar wiersza jest dosyć mały ,najlepszym sposobem wykonania tego zadania jest umieszczenie danych dla każdego wiersza tablicy w osobnej linii. Rozważmy następującą deklarację tablicy 4x4:

```
A               integer 0,1,2,3
                integer 1,0,1,1
                integer 5,7,2,2
```

```
                integer     0,0,7,6
```

Znowu asembler nie troszczy się jak podzieliliśmy linie, ale powyższe opis jest dużo łatwiejszy do identyfikacji jako tablica 4x4 jeśli zapiszemy to jako:

```
A               integer 0,1,2,3,1,0,1,1,5,7,2,2,0,0,7,6
```

Oczywiście, jeśli mamy dużą tablicę, tablicę z prawdziwie długimi wierszami lub tablicę wielowymiarową, jest mała nadzieja że wsadzisz tam coś mądrego**.**

5.6.2.4 DOSTĘP DO ELEMENTÓW WIELOWYMIAROWEJ TABLICY W JĘZYKU ASSEMBLERA

Cóż, widzieliśmy już formuły dla obliczania adresów elementów tablicy. Patrzyliśmy nawet na kod pascalowski, jaki możemy użyć przy dostępie do elementów wielowymiarowej tablicy. Teraz nadszedł czas zobaczyć jak uzyskujemy dostęp do elementów tych tablic przy użyciu języka asemblera.

Instrukcje mov, add i mul mało pracują z różnymi równaniami, które obliczają offset tablicy wielowymiarowej .Rozpatrzmy najpierw tablicę dwu wymiarową :

```
;Uwaga: rozmiar wiersza TwoD to 16 bajtów.

TwoD            integer  4 dup (8 dup (?))
i               integer  ?
j               integer  ?
                -        -
                -        -
;wykonujemy operację TwoD[i,j]:=5 używając kodu:
                mov      ax ,8              ;8 elementów w wierszu
                mul      i
                add      ax,j
                add      ax, ax             ;mnożenie przez rozmiar elementu (2)
                mov      bx, ax             ;włożenie do rejestru który używamy
                mov      TwoD [bx],5
```

Oczywiście, jeśli mamy chip 80386 (lub lepszy) możemy użyć następującego kodu:

```
                mov      eax,8
                mul      i
                add      ax, j
                mov      TwoD[eax*2],5
```

Zauważmy, że ten kod nie wymaga użycia dwóch rejestrów w trybie adresowania na 80x86.Chociaż tryb adresowania taki jak TwoD[bx][si] wygląda tak jak powinien być naturalny dostęp do dwuwymiarowych tablic, choć nie jest to cel tego trybu adresowania.

Teraz rozpatrzmy drugi przykład, który używa trójwymiarowej tablicy:

```
ThreeD          integer          4 dup (4 dup (4 dup (?)))
i               integer          ?
j               integer          ?
k               integer          ?
                -                -
                -                -
                -                -
;wykonamy operację ThreeD[i,j,k]:=1,używamy kodu:
                mov      bx, 4              ;4 elementy w kolumnie
                mov      ax,1
                mul      bx
                add      ax, j
                mul      bx                 ;4 elementy w wierszu
                add      ax, k
                add      ax,ax              ;mnożenie przez rozmiar elementu (2)
                mov      bx,ax              ;włożenie do rejestru którego używamy
                mov      ThreeD [bx],1
```

Oczywiście, jeśli mamy procesor 80386 lub lepszy, możemy to wykonać poprzez użycie następującego kodu:

```
mov     ebx ,4
mov     eax, ebx
mul     i
add     ax, j
mul     bx
add     k
mov     ThreeD[eax*2],1
```

5.6.3 STRUKTURY

Druga główną zbiorową strukturą danych jest pascalowski rekord lub struktura z C. Terminologia Pascalowska jest prawdopodobnie lepsza ,ponieważ jest tendencja do unikania zamieszania z ogólną terminologią struktur danych. Jednak, MASM używa nazwy „struktury" więc nie ma sensu odstępować od tego .Ponadto MASM używa terminu rekord do określenia jakichś drobnych różnic, kolejny powód do trzymania się jednej terminologii .

Podczas gdy tablice są homogeniczne, czyli elementy są tego samego typu, elementy w strukturze mogą być różnych typów. Tablice pozwalają nam wyselekcjonować poszczególne elementy poprzez indeks całkowity. W strukturach, musimy wybrać element (znany jako pole) poprzez nazwę.

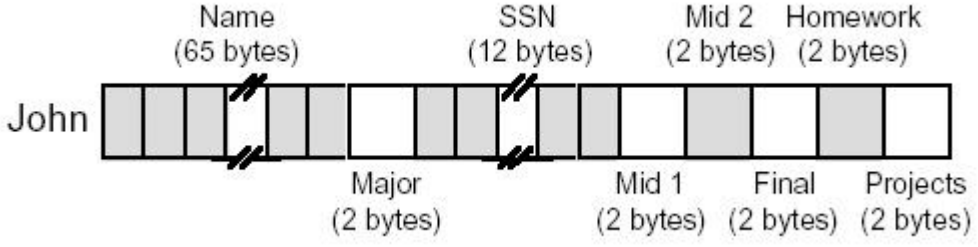
Celem struktury jest pozwolić nam na hermetyzowanie różnych, ale logicznie pokrewnych, danych wewnątrz jednego pakietu. Deklaracja pascalowskiego rekordu dla studenta jest prawdopodobnie najlepszym przykładem:

```
student = rekord
            Name:string[64];
            Major: integer;
            SSN: string[11];
            Midterm1: integer;
            Midterm2: integer;
            Final: integer;
            Homework: integer;
            Projects: integer;
end;
```

Większość pascalowskich kompilatorów alokuje każde pole rekordu w sąsiadujących komórkach pamięci. To znaczy, że Pascal zarezerwuje pierwsze 65 bajtów dla Name, następne dwa bajty zatrzyma dla Major, następne 12 bajtów dla SSN itp.

W asemblerze, możemy również stworzyć typ strukturalny używając instrukcji struct MASMa .Możemy zakodować powyższy rekord w asemblerze jak następuje:

```
student         struct
Name            char        65 dup (?)
Major           integer     ?
SSN             char        12 dup (?)
Midterm1        integer     ?
Midterm2        integer     ?
Final           integer     ?
Homework        integer     ?
Projects        integer     ?
student         ends
```



Rysunek 5.7: Przechowywanie w pamięci struktury danych Student

Zauważmy, że strukturę kończymy instrukcją ends (od end structure).Etykieta w instrukcji ends musi być taka sama jak w instrukcji struct.

Nazwy pól wewnątrz struktury muszą być unikalne. To znaczy, ta sam nazwa nie może występować dwa lub więcej razy w tej samej strukturze .jednak, wszystkie nazwy pól są lokalne w danej strukturze. Dlatego też, możemy użyć ponownie te nazwy pól gdzie indziej w programie.

Dyrektywa struct tylko definiuje typ strukturalny .Nie rezerwuje pamięci dla zmiennej strukturalnej. W rzeczywistości dla zarezerwowania pamięci musimy zadeklarować zmienną używającą nazwy struktury jako instrukcji MASMa ,np.:

```
John        student  {}
```

Nawiasy klamrowe muszą pojawić się w polu operandu. Każda wartość inicjująca musi pojawić się między nawiasami. Powyższa deklaracja alokuje pamięć jak pokazano na rysunku 5.7

Jeśli etykieta John zgadza się z adresem bazowym tej struktury, wtedy pole Name jest offsetem John+0,pole Major jest offsetem John+65,pole SSN jest offsetem John+67 itd.

Przy dostępie do elementów struktury musimy znać offset od początku struktury żądanego pola .Na przykład, pole Major zmiennej John jest offsetem 65 od adresu bazowego John .Dlatego też, możemy przechować wartość w ax w tym polu używając instrukcji mov John[65],ax. Niestety, wprowadzanie do pamięci wszystkich offsetów pól w strukturze obala potrzebę użycia tablic. W końcu jeśli mamy zająć się tymi offsetami numerycznymi dlaczego nie użyć tablicy bajtów zamiast struktury?

Cóż, jak się okazuje, MASM pozwala nam odnieść się do nazwy pola w strukturze używając tego samego mechanizmu. C i Pascal używają operatora dot (kropki).Przechowując ax w polu Major, możemy użyć mov John. Major, ax, zamiast poprzedniej instrukcji. Jest to dużo bardziej czytelne i łatwiejsze w użyciu.

Zauważmy ,że użycie operatora dot nie wprowadza nowego trybu adresowania. Instrukcja mov John. Major, ax używa trybu adresowania „tylko przemieszczenie". MASM po prostu dodaje adres bazowy John do offsetu pola Major (65) uzyskując rzeczywiste przemieszczenie dla kodowania instrukcji.

Możemy również wyspecyfikować domyślną wartość inicjującą kiedy tworzymy strukturę. W poprzednim przykładzie, pola struktury student zawierały wartości nieokreślone, wyspecyfikowane przez „?" w polu operandu każdego zadeklarowanego pola. Okazuje się, że są dwa różne sposoby dla specyfikacji wartości inicjującej dla pól struktury. Rozważmy następującą definicję „punktu" struktury danych:

```
Punkt       struct
x           word  0
y           word  0
z           word  0
Punkt       ends
```

Zawsze gdy zadeklarujemy zmienną typu punkt używając instrukcji podobnej do

```
CurPoint    Punkt   {}
```

MASM automatycznie zainicjuje zmienne CurPoint.x,CurPoint.y i CurPoint. z zerami. Układa się to świetnie w tych przypadkach gdzie nasze obiekty rozpoczynają się od tych samych wartości inicjujących. Oczywiście, możemy założyć ,że chcielibyśmy zainicjować pola X,Y i Z punktu ,ale chcemy nadać każdemu punktowi inną wartość. Jest to łatwe do osiągnięcia poprzez wyspecyfikowanie wartości inicjujących wewnątrz nawiasów:

```
Punkt1      point       {0,1,2}
Punkt2      point       {1,1,1}
Punkt3      point       {0,1,1}
```

MASM wypełni wartościami te pola w takim porządku w jakim pojawiają się one w polu operandu. Dla Punktu1 MASM zainicjuje pole X zerem, pole Y jedynką a pole Z dwójką.

Typ wartości inicjującej w polu operandu musi pasować do typu odpowiadającego mu pola w definicji struktury. Nie możemy, na przykład, wyspecyfikować stałej całkowitej dla pola real4,lub wartości większej niż 256 dla pola bajt.

MASM nie wymaga, żebyśmy inicjowali wszystkie pola w strukturze. Jeśli opuścimy jakieś pole, MASM użyje wartości domyślnej (nieokreślonej jeśli specyfikujemy „?" zamiast wartości domyślnej).

5.6.4 STRUKTURY TABLIC I TABLICE/STRUKTURY JAKO POLA STRUKTURY

Struktury mogą zawierać inne struktury lub tablice jako pola. Rozważmy następującą definicję:

```
Pixel       struct
Pt          point   {}
Color       dword   ?
```

```
Pixel       ends
```
Powyższa definicja definiuje pojedynczy punkt z 32 bitowym komponentem color. Kiedy inicjujemy obiekt typu Pixel, pierwsza inicjacja odpowiada polu Pt, nie polu współrzędnej x .**Następująca definicja jest nieprawidłowa**:
```
ThisPt      Pixel   {5,10}
```
Wartość pierwszego pola („5") nie jest obiektem typu punkt. Dlatego, asembler wygeneruje błąd kiedy napotka taką instrukcję. MASM pozwala nam zainicjować pole ThisPt używając deklaracji takiej jak następująca:
```
ThisPt      Pixel   {,10}
ThisPt      Pixel   {{},10}
ThisPt      Pixel   {{1,2,3},10}
ThisPt      Pixel   {{1,,1};,10}
```
Pierwszy i drugi z powyższych przykładów używają wartości domyślnej dla pola Pt (x=0,y=0,z=0) i ustawiają pole Color na 10.Zauważmy,że używamy nawiasów otaczających wartości inicjujące dla typu punkt w drugim, trzecim i czwartym przykładzie. Trzeci przykład inicjuje odpowiednio pola x,y i z pola Pt wartościami jeden, dwa i trzy .Ostatni przykład inicjuje pola x i z jedynkami i pozwala polu y pobrać wartość inicjującą wyspecyfikowaną przez strukturę Punkt (zero).

Dostęp do pól Pixel jest bardzo łatwy. Podobnie jak w językach wysokiego poziomu używamy jednej kropki odnosząc się do pola Pt i drugiej kropki przy dostępie do pól x,y i z punktu:
```
            mov     ax, ThisPt.Pt.X
            -
            -
            -
            mov     ThisPt.Pt.Y,0
            -
            -
            -
            mov     ThisPt.Pt.Z,d1
            -
            -
            -
            mov     ThisPt.Color, EAX
```
Możemy również zadeklarować tablice jako pola struktury. Następująca struktura tworzy typ danych zdolnych do przedstawiania obiektu z ośmioma punktami (np. sześcian):
```
Object8     struct
Pts         punkt       8 dup (?)
Color       dword       0
Object8     ends
```
Ta struktura alokuje pamięć dla ośmiu różnych punktów. Dostęp do elementów tablicy Pts wymaga znajomości rozmiaru obiektu typu punkt (pamiętamy, że musimy pomnożyć indeks tablicy przez rozmiar jednego elementu, sześć w tym szczególnym przypadku).Przypuśćmy ,na przykład, że mamy zmienną CUBE typu Object8.Możemy uzyskać dostęp do elementów Pts jak następuje:
```
; CUBE.Pts[i].X:=0;
            mov     ax ,6
            mul     i           ;sześć bajtów na element.
            mov     si, ax
            mov     CUBAEA.Pts[si].X,0
```
Jednym nieszczęśliwym aspektem tego wszystkiego jest to ,że musimy znać rozmiar każdego elementu tablicy Pts .Na szczęście MASM dostarcza operatora który oblicza rozmiar elementu tablicy (w bajtach) dla nas.

5.6.5 WSKAŹNIKI DO STRUKTUR

Podczas wykonywania naszych programów możemy odnieść się do obiektów struktury bezpośrednio lub pośrednio używając wskaźników. Kiedy używamy wskaźnika przy dostępie do pola struktury, musimy załadować jeden z rejestrów wskaźnikowych 80x86 (si, di, bx lub bp na procesorach wcześniejszych niż 80386) offsetem a es, ds., ss lub cs segmentem żądanej struktury. Przypuśćmy, że mamy następujące zmienne zadeklarowane (zakładając strukturę Object8 z poprzedniej sekcji):
```
Cube        Obkject8    {}
CubePtr     dword       Cube
```

CubePtr zawiera adres (jest to wskaźnik do) obiektu Cube .Dostęp do pola Color obiektu Cube możemy uzyskać instrukcją taką jak mov eax, Cube. Color. Kiedy uzyskujemy dostęp do pola poprzez wskaźnik musimy załadować adres obiektu do pary segemnt:rejestr wskaźnikowy ,tak jak es:bx. Instrukcja les bx,CubePtr zrobi tą sztuczkę. Po zrobieniu tego możemy uzyskać dostęp do pola obiektu Cube używając trybu adresowania disp+bx. Powstaje problem „Jak wyspecyfikować do którego pola chcemy uzyskać dostęp?" Rozważmy na krótko następujący nieprawidłowy kod:

```
les     bx,CubePtr
mov     eax, es:[bx].Color
```

Jest jeden główny problem z powyższym kodem .Ponieważ nazwy pól są lokalne w strukturze i jest możliwe użycie ponowne nazwy pola w dwóch lub więcej strukturach, jak MASM ustali który offset reprezentuje Color? Kiedy uzyskujemy dostęp do członków struktury bezpośrednio (np. mov eaz, Cube.Color) nie ma niejasności ponieważ Cube ma specyficzny typ który asembler może sprawdzić. Es:bx ,z drugiej strony, może wskazywać cokolwiek. W szczególności, może wskazywać każdą strukturę która zawiera pole Color. Więc asembler nie może, zdecydować który offset jest używany dla symbolu Color.

MASM rozwiązuje tą niejasność poprzez wymagane wyraźne określenie typu w tym przypadku. Prawdopodobnie najłatwiejszym sposobem zrobienia tego jest wyspecyfikowanie nazwy struktury jako pseudo-pola:

```
les     bx,CubePtr
mov     eax, es:[bx].Object8Color
```

Poprzez wyspecyfikowanie nazwy struktury, MASM wie która wartość offsetu jest używana dla symbolu Color.

5.10 PODSUMOWANIE

Ten rozdział przedstawia widok centralny organizacji pamięci i struktur danych 80x86.To oczywiście nie jest kompletnym kursem struktur danych. Ten rozdział omawia podstawowe i proste komponenty typów danych i jak deklarować i używać ich w programach. Mnóstwo dodatkowych informacji na temat deklarowania i używania prostych typów danych pojawi się w późniejszych rozdziałach.

Jednym z głównych problemów tego rozdziału jest omówienie jak deklarować i używać zmiennych w programach asemblerowych. W programie asemblerowym możemy łatwo tworzyć bajt, słowo, podwójne słowo i inne typy zmiennych. Takie jak skalarne typy danych wspierające boolowskie, znaki, liczby całkowite, real i inne pojedyncze typy danych znane z języków wysokiego poziomu. Zobacz:

- Deklarowanie zmiennych w programie asemblerowym
- Deklarowanie i używanie zmiennych Byte
- Deklarowanie i używanie zmiennych Word
- Deklarowanie i używanie zmiennych Dword
- Deklarowanie i używanie zmiennych Fword, Qword i Tbyte
- Deklarowanie Zmiennych zmienno przecinkowych real4,real8 i real10

Dla tego kto nie lubi używać nazw zmiennych takich jak byte, word itp. MASM pozwala tworzyć swoje własne typy nazw. Możemy je nazwać Integers zamiast Words? Żaden problem, możemy zdefiniować własne typy nazw używając instrukcji typedef .Zobacz:

- Tworzenie własnych typów nazw z TYPEDEF

Innym ważnym typem danych jest wskaźnik. Wskaźniki są niczym więcej niż adresami pamięci przechowywanymi w zmiennych (zmienna słowo lub podwójne słowo).CPU 80x86 wspiera dwa typy wskaźników - bliskie i dalekie wskaźniki. W trybie rzeczywistym, bliskie wskaźniki są długie na szesnaście bitów i zawierają offset wewnątrz znanego segmentu (zwykle w segmencie danych).Dalekie wskaźniki są długie na 32 bity i zawierają pełny adres logiczny segment:offset. Pamiętajmy, że musimy użyć jednego z rejestrów pośrednich lub trybu adresowania indeksowanego przy dostępie do danych wskazywanych przez wskaźniki. Dla tych którzy chcą tworzyć swoje własne typy wskaźnikowe (zamiast po prostu używać word lub dword dla deklaracji bliskich i dalekich wskaźników) instrukcja typedef pozwala tworzyć nazwy typów wskaźników. Zobacz:

- Typ danych wskaźnikowych

Zbiorowy typ danych jest tworzony z innego typu danych. Przykłady zbiorowych typów danych można mnożyć ,ale dwa z najbardziej popularnych zbiorowych typów danych to tablice i struktury (rekordy).Tablica jest grupą zmiennych, wszystkich tego samego typu. Program wybiera element z tablicy używając indeksu całkowitego w tej tablicy. Struktury, z drugiej strony, mogą zawierać pola których typy są różne. W programie, wybieramy żądane pole poprzez dostarczenie pola nazwy z operatorem dot. Zobacz:

- Tablice

- Tablice wielowymiarowe
- Struktury
- Tablice i Struktury i Tablice/Struktury jako Pola Struktury
- Wskaźniki na Struktury

5.11 PYTANIA

1) W jakim segmencie (8086) normalnie umiejscawiamy nasze zmienne?
2) Który segment w pliku SHELL.ASM odpowiada segmentowi zawierającemu nasze zmienne?
3) Opisz jak zadeklarować zmienne bajtowe. Daj kilka przykładów. Po co używamy zmiennych bajtowych w programie?
4) Opisz jak zadeklarować zmienne word. Daj kilka przykładów. Opisz po co używamy ich w programach
5) Powtórz pytanie 2 dla zmiennych podwójne słowo
6) Wyjaśnij cele instrukcji typedef. Daj kilka przykładów jej użycia
7) Co to jest zmienna wskaźnikowa?
8) Jak uzyskać dostęp do obiektu wskazywanego przez daleki wskaźnik. Daj przykład używając instrukcji 8086
9) Jak jest różnica między bliskim a dalekim wskaźnikiem?
10) Co to jest zbiorowy typ danych?
11) Jak deklarujemy tablice w asemblerze? Podaj kod dla następujących tablic:
    a) dwuwymiarowa tablica bajtów 4x4
    b) tablica zawierająca 128 podwójnych słów
    c) tablica zawierająca 16 słów
    d) trójwymiarowa tablica słów 4x5x5
12) Omów jak możemy uzyskać dostęp do pojedynczego elementu każdej z powyższych tablic.
13) Dostarcz kodu 80386,używając trybu adresowania indeksowego ze skalowaniem, przy uzyskaniu dostępu do elementów powyższych tablic
14) Wyjaśnij różnice pomiędzy rzędową a kolumnową organizacją elementów tablic
15) Przypuśćmy, że mamy dwu wymiarową tablicę której wartości chcemy zainicjalizować jak następuje:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Dostarcz deklaracje zmiennych do wykonania tego .Notka: Nie używaj instrukcji maszynowych 8086 dla inicjacji tablicy. Zainicjalizuj tablicę w swoim segmencie danych

```
Date=           Record
                        Month:integer;
                        Day:integer;
                        Year:integer;
                end;

Time=           Record
                        Hours:integer;
                        Minutes:integer;
                        Seconds:integer;
                end;

VideoTape =             record
                          Title:string[25];
                          ReleaseDate:Date;
                          Price:Real; (* Assume four byte reals *)




                          Length: Time;
                          Rating:char;
                        end;

  TapeLibrary : array [0..127] of VideoTape; (*This is a variable!*)
```

16) Przypuśćmy, że ES:BX wskazuje na obiekt typu VideoTape. Jaka jest instrukcja która poprawnie załaduje pole Rating do AL.