ROZDZIAŁ CZWARTY:**ROZMIESZCZENIE W PAMIĘCI I DOSTĘP**

Rozdział Pierwszy omawia podstawowe formaty danych w pamięci. Rozdział Trzeci pokazuje jak system komputerowy fizycznie organizuje te dane. Ten rozdział omawia jak CPU 80x86 uzyskują dostęp do danych w pamięci.

4.0 WSTĘP

Ten rozdział tworzy ważny pomost pomiędzy sekcją jeden i dwa (odpowiednio Organizacja Maszynowa i Podstawy Języka Asemblera).Z punktu widzenia organizacji maszynowej, ten rozdział omawia adresowanie pamięci, organizację pamięci ,tryby adresowania CPU i przedstawianie danych w pamięci .Z punktu widzenia programowania w jeżyku asemblera, rozdział ten omawia zbiór rejestrów 80x86,tryby adresowania pamięci 80x86 i złożone typy danych. Jest to rozdział kluczowy. Jeśli nie zrozumiemy materiału w tym rozdziale, będziemy mieli kłopoty ze zrozumieniem następnych rozdziałów. Dlatego też, przestudiujmy ten rozdział starannie zanim będziemy kontynuować.

Rozdział zaczyna się od omówienia rejestrów procesorów 80x86.Te procesory są wyposażone w zbiór rejestrów ogólnego przeznaczenia, rejestry segmentowe i kilku rejestrów specjalnego przeznaczenia .Pewni członkowie rodziny ,są wyposażeni w dodatkowe rejestry, chociaż typowe aplikacje ich nie używają.

Po przedstawieniu rejestrów ,rozdział opisuje organizację pamięci i segmentację w 80x86.Segmentacja jest trudną koncepcją dla wielu początkujących programistów asemblerowych. Istotnie, tekst ten stara się unikać adresowania segmentowego przez cały wstępny rozdział. Niemniej jednak. segmentacja jest potężną koncepcją która musi stać się dobrze znana komuś kto chce pisać nie trywialne programy pod 80x86.

Tryby adresowania pamięci 80x86 są, być może ,najważniejszym tematem w tym rozdziale .Jeżeli nie opanujesz całkowicie używania tych trybów adresowania ,nie będziesz mógł pisać rozsądnych programów asemblerowych .Nie przechodź dalej nim zadowalająco nie zapoznasz się z trybami adresowania. Ten rozdział omawia również rozszerzone tryby adresowania z procesora 80386 (i późniejszych). Wiedza o tych trybach adresowania nie jest tak ważna ,ale jeśli się ich nauczymy, możemy używać ich kiedy piszemy kod dla procesorów 80386 i późniejszych.

Rozdział ten również wprowadza garść instrukcji 80x86.Chociaż pięć lub więcej instrukcji tego rozdziału jest niewystarczających dla napisania prawdziwego programu asemblerowego ,dostarczają one wystarczającego zbioru instrukcji, który pozwala manipulować strukturą zmiennych i danych – temat następnego rozdziału.

4.1 CPU 80x86: SPOJRZENIE PROGRAMISTY

Teraz ,jest czas aby omówić kilka prawdziwych procesorów:8088/8086/80188,80286 i 80386/80486/80586/Pentium.Rozdział Trzeci zajmował się dużo sprzętowymi aspektami systemu komputerowego. Te komponenty sprzętowe wpływają na sposób w jaki piszemy programy ,lecz dla CPU jest dużo więcej rzeczy niż tylko cykle magistrali i potoki. Jest czas spojrzeć na te komponenty CPU które są bardziej widoczne dla nas, programistów asemblerowych.

Najbardziej widocznymi komponentami CPU jest zbiór rejestrów .Podobnie jak nasze hipotetyczne procesory ,chipy 80x86 mają zbiór rejestrów zintegrowanych na płycie .Zbiór rejestrów dla każdego procesora w rodzinie 80x86 jest nadzbiorem tamtych poprzednich CPU. Najlepszym miejsce do rozpoczęcia jest zbiór rejestrów procesorów 8088,8086,80188 i 80186 ponieważ te cztery procesory mają takie same rejestry. Przy ich omawianiu ,termin „8086” będzie sugerował każdy z tych czterech CPU.

Projektanci Intela zaklasyfikowali rejestry 8086 do trzech kategorii: rejestry ogólnego przeznaczenia, rejestry segmentowe i różnorodne rejestry .Rejestry ogólnego przeznaczenia są tymi które mogą stać się operandami arytmetycznych ,logicznych i pokrewnych instrukcji. Chociaż te rejestry są „ogólnego przeznaczenia", każdy z nich ma swojej własne przeznaczenie. Intel używa terminu „ogólnego przeznaczenia" luźno.8086 używa rejestrów segmentowych przy dostępie do bloków pamięci nazywanych, dość niespodziewanie, segmentami. Zobacz „Segmenty w 80x86" po większą ilość szczegółów dotyczącą natury rejestrów segmentowych. Ostatnią klasą rejestrów 8086 są rejestry. Są dwa specjalne rejestry w tej grupie które omówimy wkrótce.

4.1.1 REJESTRY OGÓLNEGO PRZEZNACZENIA 8086

Jest osiem 16 bitowych rejestrów ogólnego przeznaczenia w 8086: ax,bx,cx,dx,si,di,bp i sp .Możemy używać wielu z tych rejestrów zamiennie w obliczeniach, wiele instrukcji pracuje bardziej wydajnie lub całkowicie wymaga specyficznego rejestru z tej grupy. Tyle o ogólnym przeznaczeniu.

Rejestr **ax** (akumulator) występuje wszędzie tam gdzie mają miejsce arytmetyczne lub logiczne obliczenia. Chociaż możemy wykonywać operacje arytmetyczne i logiczne na innych rejestrach ,często bardziej wydajne jest używanie rejestru ax do takich obliczeń. Rejestr **bx** (bazowy) ma również kilka specjalnych przeznaczeń .Jest on powszechnie używany do przechowywani adresu pośredniego, podobnie jak rejestr bx z procesorów x86.Rejestr **cx** (licznik),jak wskazuje jego nazwa wskazuje służy do obliczeń **.**Często będziemy go używać do zliczania iteracji w pętli lub specyfikowania liczby znaków w łańcuchu. Rejestr **dx** (danych) ma dwa specjalne przeznaczenia: przechowuje przepełnienie z pewnych arytmetycznych operacji i przechowuje adresy I/O kiedy uzyskujemy dostęp do danych na szynie I/O w procesorze 80x86.

Rejestry **si** i **di** (indeks źródłowy i indeks przeznaczenia) również mają kilka specjalnych przeznaczeń. Możemy używać tych rejestrów jako wskaźników (podobnie jak rejestr bx) do pośredniego dostępu do pamięci .Będziemy również używać tych rejestrów z instrukcjami łańcuchowymi 8086 kiedy przetwarzamy łańcuchy znaków.

Rejestr **bp** (wskaźnik bazowy) jest podobny do rejestru bx. Ogólnie rzecz biorąc, będziemy używać tego rejestru przy uzyskaniu dostępu do parametrów i zmiennych lokalnych w procedurze.

Rejestr **sp** (wskaźnik stosu) ma bardzo specjalne znaczenie – utrzymuje stos programu. Normalnie, nie będziemy używać tego rejestru dla obliczeń arytmetycznych. Właściwe operacje większości programów zależą od ostrożnego używania tego rejestru.

Poza tymi ośmioma 16 bitowymi rejestrami ,CPU 8086 ma również 8 ośmiobitowych rejestrów .Intel nazywa te rejestry al.,ah,bl,bh,cl,ch,dl i dh. Prawdopodobnie zauważyłeś podobieństwa miedzy tymi nazwami a nazwami kilku rejestrów 16 bitowych (ax,bx,cx i dx)Te ośmiobitowe rejestry nie są niezależnymi rejestrami al reprezentuje „mniej znaczący bajt ax",ah reprezentuje „bardziej znaczący bajt ax". Nazwy innych ośmiobitowych rejestrów znaczą to samo dla rejestrów bx,cx i dx. Rysunek 4.1 pokazuje zbiór rejestrów ogólnego przeznaczenia.

Zauważ, że ośmiobitowe rejestry nie tworzą niezależnego zbioru rejestrów. modyfikowanie al. zmieni wartość ax; więc zmodyfikuje ah .Wartość al. dokładnie odpowiada bitom od zera do siedem ax .Wartość ah odpowiada bitom od osiem do piętnaście ax .Dlatego też modyfikacja al lub ah zmodyfikuje wartość ax .Podobnie, zmodyfikowanie ax zmieni oba al. i ah .Zauważ, jednak ,że zmieniający się al. nie wpłynie na wartość ah i vice versa .Odnosi się to również do bx/bl/bh,cx.cl.ch i dx,dl,dh.

Rejestry si,di,bp i sp są tylko rejestrami 16 bitowymi .Nie ma sposobu na bezpośredni dostęp do pojedynczych bajtów w tych rejestrach tak jak możemy uzyskać dostęp do młodszych i starszych bajtów rejestrów ax,bx,cx i dx.

Rysunek 4.1 Zbiór rejestrów 8086

4.1.2 REJESTRY SEGMENTOWE 8086

8086 ma cztery specjalne rejestry segmentowe: **cs**,**ds**,**es** i **ss** .Oznaczają one odpowiednio Segment Kodu, Segment Danych, Segment Extra i Segment Stosu. Te rejestry wszystkie są szerokie na 16 bitów .Zajmują się one wyselekcjonowywaniem bloków (segmentów) pamięci głównej. Rejestr segmentowy (np. cs) wskazuje początek segmentu w pamięci.

Segmenty pamięci w 8086 nie mogą być dłuższe niż 65,536 bajtów .Jest to niesławne „ograniczenie segmentu do 64K" przeszkadzające wielu programistom. Zobaczymy później kilka problemów z ograniczeniem 64K,i kilka rozwiązań tego problemu.

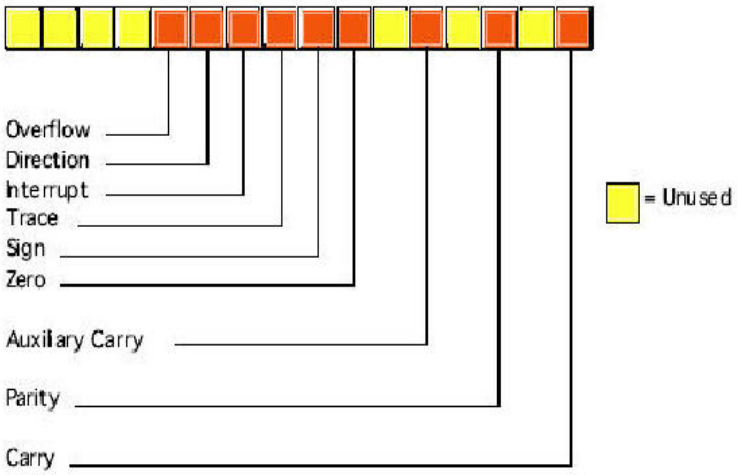
Rejestr cs wskazuje segment zawierający obecnie wykonywane instrukcje maszynowe. Zauważmy ,że pomimo ograniczenia segmentu do 64K,programy 8086 mogą być dłuższe niż 64K.Po prostu potrzebujemy większą ilość segmentów kodu w pamięci. Ponieważ możemy zmieniać wartość rejestru cs ,możemy przejść do nowego segmentu kodu kiedy chcemy wykonać kod tam umieszczony.

Rejestr segmentu danych, ds., ogólnie rzecz biorąc, wskazuje na globalne zmienne programu .Znowu jesteśmy ograniczeni 65,536 bajtami danych w segmencie danych; ale zawsze możemy zmienić wartość rejestru ds. aby uzyskać dostęp do dodatkowych danych w innym segmencie.

Rejestr ekstra segmentu, es ,jest dokładnie tym - rejestrem ekstra segmentu. Programy 8086 często używają tego rejestru segmentowego aby uzyskać dostęp do segmentów kiedy trudno jest lub jest niemożliwe modyfikowanie innych rejestrów segmentowych.

Rejestr ss wskazuje segment zawierający stos 8086.Stos występuje wtedy kiedy przechowujemy ważne informacje stanu maszyny, powrót z podprogramu adresowania, parametry procedury i lokalne zmienne. Ogólnie rzecz biorąc, nie modyfikujemy rejestru segmentu stosu ponieważ zbyt wiele rzeczy w systemie zależy od tego.

Chociaż jest teoretycznie możliwe przechowywać dane w rejestrach segmentowych, nie jest to dobry pomysł. Rejestry segmentowe mają bardzo specjalne przeznaczenie – wskazywanie dostępnych bloków pamięci. Próby używania tych rejestrów do innych celów mogą w wyniku dać wiele zmartwienia ,zwłaszcza jeśli zamierzamy przejść na lepszy CPU np. 80386



Rysunek 4.2 : Rejestr Flag

4.1.3 REJESTRY SPECJALNEGO PRZEZNACZENIA 8086

Są dwa rejestry specjalnego przeznaczenia w CPU 8086: wskaźnik instrukcji (IP) i rejestr flag .Nie uzyskamy dostępu do tych rejestrów w ten sam sposób jak do innych rejestrów 8086.Zamiast tego ,CPU manipuluje tymi rejestrami bezpośrednio.

Rejestr IP jest odpowiednikiem rejestru IP procesorów x86 – zawiera adres obecnie wykonywanej instrukcji .Jest to 16 bitowy rejestr ,który zapewnia wskaźnik do bieżącego segmentu kodu (16 bitów pozwala wybrać jeden z 65,536 różnych komórek pamięci).Wrócimy do tego rejestru kiedy będziemy omawiać później transmisję sterowania instrukcjami.

Rejestr flag nie jest podobny do innych rejestrów na 8086.Inne rejestry przechowują ośmio- lub 16 bitowe wartości. Rejestr flag jest po prostu zbiorem kompilacyjnych jednobitowych wartości które pomagają określić bieżący stan procesora .Chociaż rejestr flag jest szeroki na 16 bitów,8086 używa tylko dziewięciu z tych

bitów. Z tych flag ,czterech flag używamy cały czas: znacznik zera, znacznik przeniesienia, znacznik znaku i znacznik nadmiaru. Te flagi są kodami stanu 8086.Rejestr flag jest pokazany na rysunku 4.2

4.1.4 REJESTRY 80286

Mikroprocesor 80286 dodaje jedną ważną z programistycznego punktu widzenia cechę do 80286-operacje trybu chronionego. Ten tekst nie obejmuje operacji trybu chronionego dla 80286 z różnych przyczyn. Po pierwsze ,tryb chroniony 80286 był kiepsko zaprojektowany. Po drugie jest on interesujący tylko dla programistów piszących swoje własne systemy operacyjne lub nisko poziomowe programy systemowe dla takiego systemu operacyjnego .Nawet jeśli piszemy oprogramowanie dla trybu chronionego systemu operacyjnego takiego jak UNIX czy OS/2 nie będziemy używać cech trybu chronionego 80286.Pomimo to jest wart zachodu do wskazywania ekstra rejestrów i stanu flag obecnych w 80286 właśnie w przypadku natknięcia się na nie.

Są trzy dodatkowe bity obecne w rejestrze flag 80286.Poziom Uprzywilejowania I/O to wartość dwóch bitów (bity 12 i 13).Specyfikują jeden z czterech uprzywilejowanych poziomów potrzebnych do wykonania operacji I/O. Te dwa bity generalnie zawierają 00b kiedy użytkujemy tryb rzeczywisty na 80286 (tryb emulowany na 8086).Flaga NT (nested task – znacznik zagnieżdżonego zadania) steruje operacją instrukcji powrotu z przerwania (IRET).Normalnie NT ma zero dla programu w trybie rzeczywistym.

Poza tymi ekstra bitami w rejestrze flag,80286 ma również pięć dodatkowych rejestrów używanych przez system operacyjny do wspomagania zarządzania pamięcią i przetwarzanie wieloprogramowe :rejestr stanu (msw),rejestr globalnej tablicy deskryptorów (gdtr)rejestr lokalnej tablicy deskryptorów(ldtr),rejestr tablicy deskryptorów przerwań(idtr) i rejestr stanu zadania (tr)

Przy używaniu typowej aplikacji dla trybu chronionego na 80286,mamy dostęp do więcej niż jednego megabajtu RAMu .jednakże 80286 jest praktycznie przestarzały a jest lepszy sposób dostępu do większej ilości pamięci na późniejszych procesorach, programiści rzadko używają tej formy trybu chronionego.

4.1.5 REJESTRY 80386/80486

Procesor 80386 radykalnie rozszerzył zbiór instrukcji 8086.Dodatkowo wszystkie rejestry w 80286 (zatem i 8086),80386 dodał kilka nowych rejestrów i rozszerzył definicję istniejących rejestrów.80486 nie dodał żadnych nowych rejestrów do podstawowego zbioru rejestrów 80386,ale zdefiniował kilka bitów w kilku rejestrach niezdefiniowanych przez 80386.

Bardzo ważną zmiana, z punktu widzenia programisty, było wprowadzenie w 80386 zbioru 32 bitowych rejestrów .Ax,bx,cx,dx,si,di,bp,sp, flagi i rejestr ip ,zostały wszystkie rozszerzone do 32 bitów.80386 nazwał te 32 bitowe wersje eax,ebx,ecx,ex,esi,edi,ebp,esp, eflagi i eip w odróżnieniu ich od ich 16 bitowych wersji (które są jeszcze dostępne w 80386) .Poza 32 bitowymi rejestrami,80386 również wprowadził dwa nowe 16 bitowe rejestry segmentowe fs i gs, które pozwalają programiście jednocześnie uzyskać dostęp do sześciu różnych segmentów w pamięci bez ładowania rejestru segmentu. Zauważ ,że wszystkie rejestry segmentowe w 80386 są 16 bitowe.80386 nie rozszerzyły rejestru segmentów do 32 bitów ,jak zrobiły to z innymi rejestrami.

80386 nie zrobił żadnych zmian w bitach w rejestrze flag. Zamiast tego rozszerzył rejestr flag do 32 bitów (rejestr eflag) i zdefiniował bity 16 i 17.Bit 16 jest znacznikiem wznowienia(RF) i używanym w zbiorze rejestrów uruchomieniowych .Bit 17 jest znacznikiem trybu V86(VM).który określa czy procesor pracuje w trybie wirtualnym V86 (symulowanym w 8086) lub standardowym trybie chronionym.80486 dodaje trzeci bit do rejestru eflag na pozycji 18 - flaga stanu wyrównania .Razem z rejestrem sterującym zero (CR0) w 80486,flaga ta wymusza pułapkę (przerwanie programu) kiedy procesor uzyskuje dostęp do nie wyrównanych danych.(np. słowo spod nieparzystego adresu lub podwójne słowo spod adresu który nie dzieli się przez cztery)

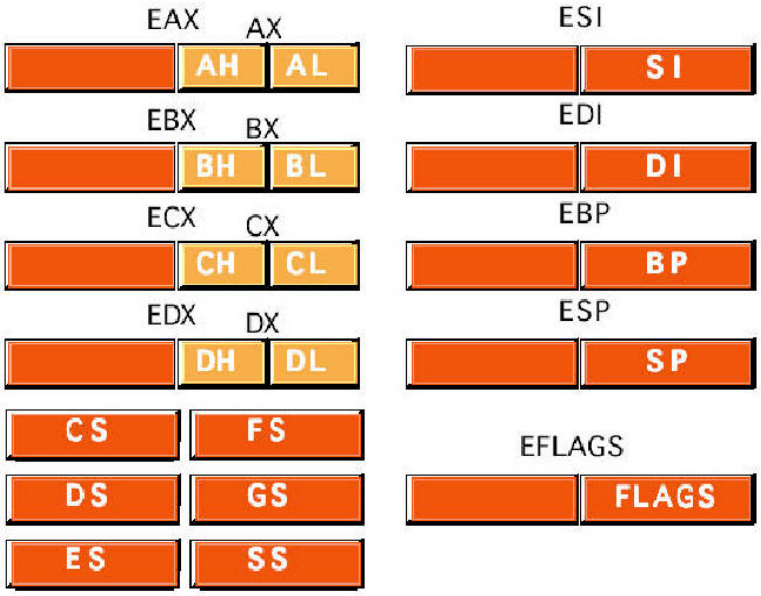
80386 dodał cztery rejestry sterujące:CR0-CR3.rejestry te rozszerzają rejestr msw 80286 (80386 emuluje rejestr msw z 80286 dla kompatybilności ,ale informacja w rzeczywistości pojawia się w rejestrach CRx)W 80386 i 80486 te rejestry sterują funkcjami takimi jak zarządzanie pamięcią stronicowaną ,włączanie/wyłączanie operacji pamięci podręcznej (tylko 80486),operacji trybu chronionego i innych.

80386/486 dodają również osiem rejestrów uruchomieniowych .Program uruchomieniowy taki jak Microsoft Codeview lub Turbo Debugger mogą używać tych rejestrów do ustawiania punktów kontrolnych ,kiedy próbujemy zlokalizować błąd wewnątrz programu. Kiedy nie będziemy używać tych rejestrów w programach użytkowych ,często odkryjemy ,że używanie takiego debuggera zredukuje czas potrzebny do wyeliminowania błędów z naszych programów. Oczywiście, debugger który uzyska dostęp do tych rejestrów będzie funkcjonował właściwie na procesorach 80386 lub późniejszych.

Ostatecznie, procesory 80386/486 dodają zbiór rejestrów testujących system ,które testują stosowne operacje procesora kiedy system jest włączany. Jest prawdopodobne, że Intel dołoży te rejestry do chipu co pozwoli testować bezpośrednio po produkcji ,ale projektanci systemu mogą wykorzystać te rejestry do robienia testów po włączeniu zasilania.

Przeważnie programiści asemblerowi nie muszą martwić się ekstra rejestrami dodanymi do procesorów 80386/80486/Pentium.Jednakże 32 bitowe rozszerzenie i ekstra rejestry segmentowe są całkiem

użyteczne. Dla programów użytkowych ,model programowania dla 80386/80486/Pentium wygląda jak ten pokazany na rysunku 4.3



Rysunek 4.3 Rejestry 80386 (Dostępne programiście asemblerowemu)

4.2 FIZYCZNA ORGANIZACJA PAMIĘCI 80x86

Rozdział Trzeci omawiał podstawową organizację Architektury Von Neumanna (VNA) systemów komputerowych. W typowej maszynie VNA,CPU łączy pamięć przez magistrale.80x86 wybiera kilka szczególnych elementów pamięci używając liczb binarnych na magistrali adresowej. Innym sposobem obejrzenia pamięci jest tablica bajtów .Pascalowska struktura danych, która z grubsza pokrywa się z pamięcią będzie wyglądać tak:

Memory : array [0..MaxRAM] of byte;

Wartość na magistrali adresowej odpowiada indeksowi dostarczonemu do tablicy. Np. zapisaniu danych do pamięci odpowiada :

Memory [adres] :=Wartość_Do_Zapisania;

Odczytaniu danych z pamięci odpowiada :

Wartość_Odczytana := Memory [adres];

Różne CPU mają różne magistrale adresowe które sterują maksymalną liczbą elementów w tablicy pamięci (zobacz „Magistrala Adresowa").Jednakże, bez względu na liczbę linii adresowych na magistrali ,większość komputerów nie ma jednego bajtu pamięci dla każdej adresowalnej lokacji .Na przykład, procesor 80386ma 32 linie adresowe pozwalające zwiększyć pamięć do czterech gigabajtów. Bardzo mało systemów 80386 w rzeczywistości ma cztery gigabajty .Zazwyczaj, mamy od jednego do 256 megabajtów w systemach opartych 0 80x86.

Pierwszy megabajt pamięci ,od adresu zero do 0FFFFFh jest specjalny w 80x86.Odpowiada to całej przestrzeni adresowej mikroprocesorów 8088,8086,80186 i 80188.Większość programów DOS ogranicza swoje programy i adresy danych do lokacji w tym zakresie. Adresy ograniczone do tego zakresu są nazywane adresami rzeczywistymi po trybie rzeczywistym 80x86.

4.3 SEGMENTY W 80x86

Nie możemy omawiać adresowania pamięci w rodzinie procesorów 80x86 bez omówienia najpierw segmentacji. Pomiędzy innymi rzeczami, segmentacja dostarcza silnych mechanizmów zarządzania pamięcią. Pozwala to programistom dzielić swoje programy na moduły które działają niezależnie jeden od drugiego. Segmenty dostarczają sposobu łatwej implementacji programów zorientowanych obiektowo. Segmenty pozwalają dwóm procesom łatwo dzielić dane. W sumie, segmentacja jest rzeczywiście zgrabną cechą. Z drugiej strony, jeśli spytamy programistów co myślą o segmentacji, co najmniej dziewięciu na dziesięciu stwierdzi ,że to straszne. Dlaczego taka odpowiedź?

Cóż, okazuje się,że segmentacja dostarcza drugiej zmyślnej cechy: pozwala nam rozszerzyć adresowalność procesora. W przypadku 80x86,segmentacja pozwoliła projektantom Intela rozszerzyć

pamięć adresowalną z 64K do jednego megabajta. Eee ..nieźle brzmi. Dlaczego wszyscy się skarżą? Cóż, krótka lekcja historii pozwoli zrozumieć co stało się złego.

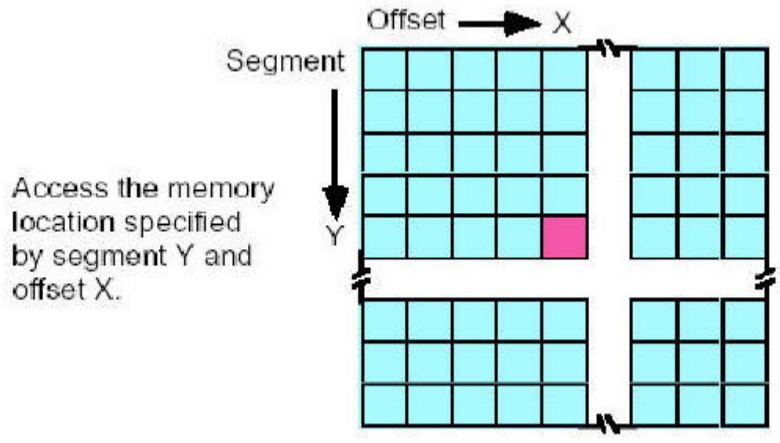
W 1976 roku ,kiedy Intel zaczął projektować procesor 8086,pamięć była bardzo droga .Komputery osobiste ,takie które były w tym czasie ,typowo miały cztery tysiące bajtów pamięci. Nawet kiedy IBM wprowadził PC pięć lat później,64K było jeszcze odpowiednią pamięcią. Jeden megabajt był olbrzymią ilością .Projektanci Intela uważali, że 64K pamięci pozostanie dużą wartością przez cały czas życia procesora 8086.Pomyłką jaką zrobili ,było kompletne przecenienie czasu życia procesora 8086.Doszli do wniosku, że będzie to około pięciu lat ,podobnie jak wcześniej procesora 8080.Zaplanowali mnóstwo innych procesorów przez ten czas, a „86" nie był przyrostkiem w nazwie każdego z nich .Intel doszedł do wniosku ,że będą one grupą .Z pewnością jeden megabajt byłby więcej niż wystarczający do momentu aż nie wyszliby z czymś lepszym.

Niestety ,Intel nie liczył na IPM PC i ogromną ilość oprogramowania pojawiającego się dla niego. W 1983 roku, było jasne, że Intel nie może porzucić architektury 80x86.Utknęli na niej ,ale do tego czasu ludzie zaczęli protestować przeciwko jedno megabajtowemu ograniczeniu 8086.Więc Intel dał nam 80286.Ten procesor mógł adresować do 16 megabajtów pamięci. Z pewnością więcej niż dość. Problemem było tylko to,że całe cudowne oprogramowanie napisane dla IBM PC było napisane w taki sposób ,że nie mogło wykorzystywać żadnej pamięci poza jeden megabajt.

Okazało się,że maksymalna ilość adresowalnej pamięci nie jest dla wszystkich główną dolegliwością. Prawdziwym problemem było to,że 8086 był procesorem 16 bitowym, z 16 bitowymi rejestrami i 16 bitowymi adresami. To ograniczało procesor do adresowania 64K kawałków pamięci. Bystrzachy z Intela użyli rozszerzonych segmentów do jednego megabajta, ale adresowanie więcej niż 64K w jednym czasie stanowiło wysiłek. Adresowanie więcej niż 256K w jednym czasie stanowiło duży wysiłek.

Pomimo tego co usłyszeliśmy ,segmentacja nie jest zła. Faktycznie ,jest to naprawdę wielki program zarządzania pamięcią. Co jest złego w tym ,że Intel w 1976 zaimplementował segmentację ciągle używaną dzisiaj .Nie możemy winić Intela za to,że – przenieśli problem z 80 na 80386.Prawdziwym sprawcą jest MS-DOS, który wymusił na programistach kontynuowania używanego od 17976 roku stylu segmentacji. Na szczęście nowsze systemy operacyjne takie jak Linux, UNIX ,Windows 9x,Windows NT i OS/2 nie cierpią z tego powodu tak jak MS-DOS. Ponadto użytkownicy chętnie przechodzą na nowsze systemy operacyjne ,więc programiści mogą wykorzystywać nowe cechy rodziny 80x86.

Odłóżmy lekcję historii na bok ,jest to prawdopodobnie dobra myśl rozpracować czym jest segmentacja. Rozważmy bieżący widok pamięci :wygląda ona jak liniowa tablica bajtów .Pojedynczy indeks (adres) wybiera jakiś szczególny bajt z tej tablicy .Nazwijmy ten typ adresowania liniowym lub płaskim .Adresowanie segmentowe używa dwóch komponentów do wyspecyfikowania komórki pamięci: wartość segmentu i offset (przesuniecie) wewnątrz tego segmentu. Idealnie ,wartości segmentu i offsetu są od siebie niezależne .Najlepszym sposobem opisu



Rysunek 4.4 Adresowanie segmentowe jako dwu wymiarowy proces

adresowania segmentowego jest dwu wymiarowa tablica .Segment dostarcza jednego indeksu do tablicy ,offset dostarcza drugiego (zobacz rysunek 4.4)

Teraz możesz się zastanawiać „Dlaczego robimy ten proces bardziej złożonym?" Adresowanie liniowe wydaje się w działaniu świetny, dlaczego zawracać sobie głowę tym dwuwymiarowym schematem adresowania? Cóż ,rozważymy sposób napisania typowego programu .Gdy napiszemy ,powiedzmy, procedurę SIN(X) i potrzebujemy jakichś zmiennych czasowych, prawdopodobnie nie użyjemy zmiennych globalnych .Zamiast tego użyjemy zmiennych lokalnych wewnątrz funkcji SIN(X).W sensie ogólnym, jest

to jedna z cech którą oferuje segmentacja – umiejętność przyłączenia bloku zmiennych (segmentu) do konkretnej części kodu. Możemy, na przykład, mieć segment zawierający lokalne zmienne dla SIN, segment dla SQRT, segment dla DRAW-Window itp. .Ponieważ zmienne dla SIN pojawiają się w segmencie dla SIN, jest mniej prawdopodobne, że nasza procedura SIN wpłynie na zmienne należące do procedury SQRT. Istotnie, w 80286 i późniejszych operujących w trybie chronionym ,CPU może uniemożliwić jednemu programowi przypadkowo zmodyfikować zmienne w różnych segmentach.

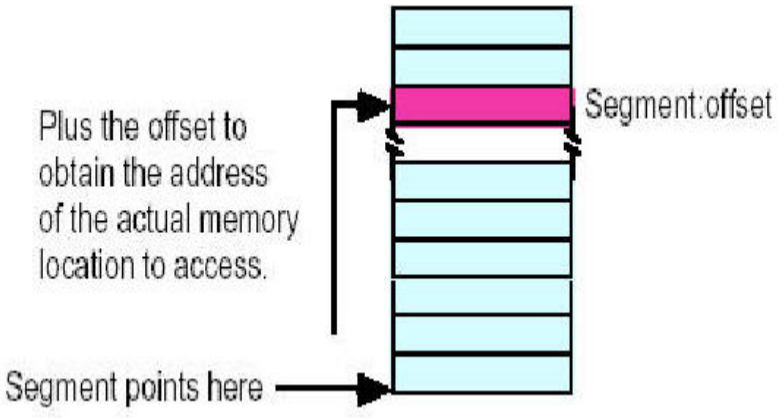
Pełen adres segmentowany zawiera część segmentową i część offsetową .W tym tekście będziemy pisać adres segmentowy jako segment:offset. W 8086 przez 80286 te dwie wartości są 16 bitowymi stałymi. W 80386 i późniejszych offset może być 16 lub 32 bitową stałą.

Rozmiar offsetu ograniczony jest do maksymalnego rozmiaru segmentu. W 8086 z 16 bitowym offsetem ,segment może być nie dłuższy niż 64K;może być mniejszy ale nigdy dłuższy.80386 i późniejsze procesory pozwalają 32 bitowym offsetom pracować z segmentami tak długimi jak cztery gigabajty.

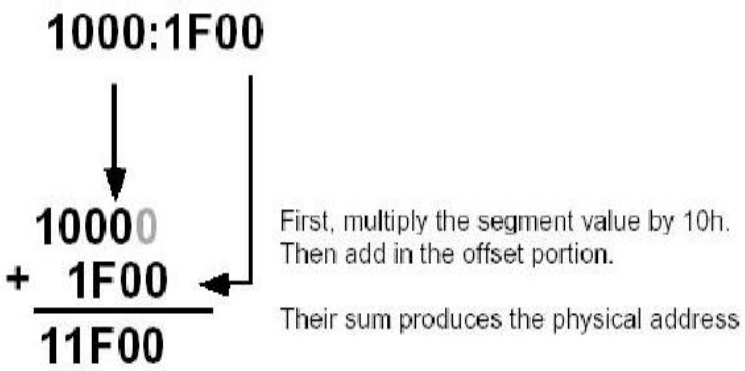
Części segmentu są 16 bitowe na wszystkich procesorach 80x86.Pozwala to pojedynczym programom mieć 65,536 różnych segmentów w programie. Większość programów ma mniej niż 16 segmentów (lub coś koło tego) więc nie jest to praktyczne ograniczenie.

Oczywiście, pomimo faktu ,że rodzina 80x86 używa adresowania segmentowego, rzeczywista (fizyczna) pamięć podłączona do CPU jest liniową tablicą bajtów. Jest funkcja która konwertuje wartość segmentu do adresu pamięci fizycznej. Procesor wtedy dodaje offset do tego fizycznego adresu uzyskując rzeczywisty adres danych w pamięci. W tym tekście będziemy się odnosić do adresów w naszych programach jako adresów segmentowych lub adresów logicznych. Rzeczywisty adres liniowy który pojawia się na magistrali adresowej jest adresem fizycznym (zobacz rysunek 4.4).

W 8086,8088,80186 i 80188 (i inne procesory operujące w trybie rzeczywistym),funkcja która odwzorowuje segment do adresu fizycznego jest bardzo prosta. CPU mnoży wartość segmentu przez szesnaście (10h) i dodaje część offsetową. Na przykład, rozważmy adres segmentowy: 1000:1F00..Konwertując go na adres fizyczny ,mnożymy wartość segmentu (1000h) przez



Rysunek 4.5 Adresowanie segmentowe w pamięci fizycznej



Rysunek 4.6 Konwertowanie adresu logicznego na adres fizyczny

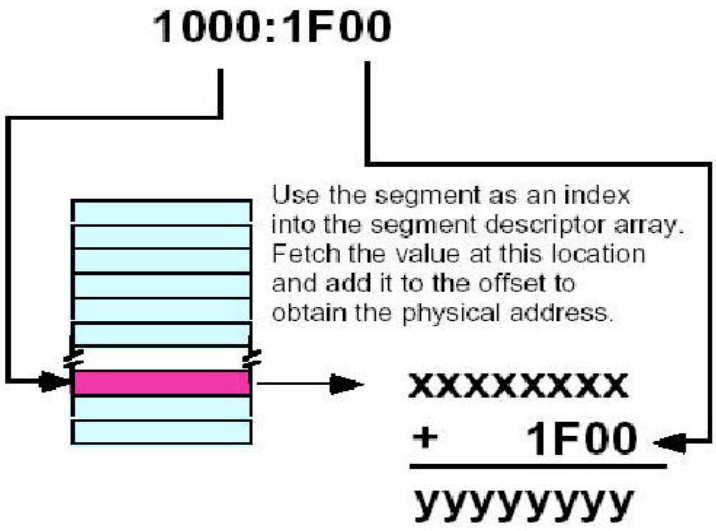
szesnaście.. Mnożenie przez podstawę systemu jest bardzo łatwe .Najpierw dołączamy zero na końcu liczby. Po dodaniu zera do 1000h otrzymujemy 10000h.Dodajemy 1F00h do tego i otrzymujemy 11F00h.Więc 11F00h jest fizycznym adresem który odpowiada adresowi segmentowemu 1000:1F00h (zobacz rysunek 4.4)

Ostrzeżenie: Bardzo popularnym błędem ludzi robionym, kiedy wykonują te obliczenia jest zapominanie ,że pracują w systemie heksadecymalnym nie dziesiętnym**.** To zaskakujące widzieć jak dużo ludzi dodaje 9+1 i otrzymuje 10h zamiast prawidłowej odpowiedzi 0Ah.

Intel, kiedy zaprojektował 80286 i późniejsze procesory, nie rozszerzył adresowania przez dodanie więcej bitów do rejestrów segmentowych. Zamiast tego, zmienili funkcję CPU używaną do konwersji adresu logicznego na adres fizyczny. Jeśli piszemy kod który zależy od funkcji „mnożenia przez 16 i dodanie offsetu" nasz program będzie mógł pracować tylko na procesorach 80x86 operujących w trybie rzeczywistym i będziemy ograniczeni do pamięci jednomegabajtowej.

W procesorach 80286 i późniejszych ,Intel wprowadził segment trybu chronionego .Wśród innych zmian, Intel zupełnie zreformował algorytm dla odwzorowywania segmentu w przestrzeni adresów liniowych .Zamiast używania funkcji (takiej jak mnożenie wartości segmentu przez 10h),tryb chroniony procesorów używa tablicy odwzorowań do obliczenia adresu fizycznego. W trybie chronionym,80286 i późniejsze procesory używają wartości segmentu jako indeksu wewnątrz tablicy .Zawartość wyselekcjonowanego elementu tablicy dostarcza (między innymi) adres startowy dla segmentu. CPU dodaje tą wartość do offsetu uzyskując adres fizyczny.(zobacz rysunek 4.4).

Zauważ ,że nasze aplikacje nie mogą bezpośrednio modyfikować tablicy deskryptora segmentów .Tryb chroniony systemów operacyjnych (UNIX,Linux,Windows,OS/2,itp.) posługują się tą operacją.



Rysunek 4.7 Konwertowanie adresu logicznego na fizyczny w trybie chronionym

Najlepsze programy nigdy nie zakładają ,że segment jest ulokowany w szczególnym miejscu pamięci. Powinniśmy zostawić to systemowi operacyjnemu, który umieści nasz program w pamięci i nie generować żadnego własnego adresu segmentowego.

4.4 ZNORMALIZOWANE ADRESY W 80x86

Kiedy pracujemy w trybie rzeczywistym, występuje interesujący problem. Możemy odnosić się do pojedynczego obiektu w pamięci używając kilku różnych adresów. Rozważmy adres z poprzedniego przykładu,1000:1F00h.Jest kilka różnych adresów pamięci, które odnoszą się do tego samego adresu fizycznego. Na przykład,11F0:0,1100:F00 a nawet 1080:1700 wszystkie odpowiadają adresowi fizycznemu 11F00h.Kiedy pracujemy z pewnymi typami danych a zwłaszcza kiedy porównujemy wskaźniki, jest dogodnie jeśli adresy segmentowe wskazują różne obiekty w pamięci kiedy ich reprezentacje bitowe są różne. Wyraźnie nie jest to przypadek w trybie rzeczywistym procesorów 80x86.

Na szczęście jest łatwy sposób uniknięcia tego problemu .Jeśli musimy porównać dwa adresy dla (nie) równości ,możemy użyć adresów znormalizowanych. Adresy znormalizowane przyjmują specjalną postać ,przez co wszystkie są unikalne .To znaczy ,o ile dwie znormalizowane wartości segmentów nie są dokładnie takie same, nie wskazują na ten sam obiekt w pamięci.

Jest wiele różnych sposobów (16 faktycznie) do stworzenia adresów znormalizowanych. Przez konwencję ,większość programistów (i języków wysokiego poziomu) definiuje adres znormalizowany jak następuje:

- Część segmentowa adresu może być każą 16 bitową wartością
- Część offsetowa musi być wartością z zakresu 0..0Fh

Wskaźniki znormalizowane ,które przyjmują taką formę są bardzo łatwo konwertowane na adres fizyczny.. Wszystko co musimy zrobić to dołączyć pojedynczą cyfrę heksadecymalną z offsetu do wartości segmentu. Postać znormalizowana z 1000:1F00 to 11F0:0.Możemy otrzymać adres fizyczny poprzez dodanie offsetu (zero) na końcu 11F0 otrzymując 11F00.

Jest to bardzo łatwy sposób konwersji dowolnej wartości segmentowej do adresu znormalizowanego. Po pierwsze konwertujemy nasz adres segmentowy do adresu fizycznego używając funkcji „mnożenie przez 16 i dodania w offsecie" Potem dajemy dwukropek między ostatnie dwie cyfry pięciocyfrowego wyniku:

1000:1F00 =>11F00=>11F0:0

Zauważ ,że to omówienie dotyczy tylko procesorów 80x86 operujących w trybie rzeczywistym. W trybie chronionym nie ma bezpośredniej odpowiedniości między adresami segmentowymi a adresami fizycznymi więc ta technika nie działa .Jednakże, ten tekst zajmuje się głównie programami, które pracują w trybie rzeczywistym, więc znormalizowane wskaźniki pojawiają się w całym tekście.

4.5 REJESTRY SEGMENTOWE W 80x86

Kiedy Intel projektował 8086 w 1976 roku ,pamięć była cenną rzeczą .Zaprojektowali swój zbiór instrukcji tak,żeby każda instrukcja używała tka mało bajtów jak to możliwe. To uczyniło programy mniejszymi więc systemy komputerowe stosowały procesory Intelowskie używające mniej pamięci. Jako takie ,te systemy komputerowe były tańsze do wytworzenia. Oczywiście, koszt pamięci spadał do punktu gdzie nie ma powodu do martwienia się o nią jak było kiedyś**.** Jednej rzeczy chciał uniknąć Intel dołączając 32 bitowy adres (segment:offset) do końca instrukcji które odnoszą się do pamięci .Chcieli zredukować to do 16 bitów (tylko offset) przez dokonanie pewnych założeń o tym do których segmentów w pamięci mogą uzyskać dostęp instrukcje.

Procesory 8086 do 80286 mają cztery rejestry segmentowe ;cs,ds.,ss i es.80386 i późniejsze procesory mają te rejestry segmentowe plus fs i gs. Rejestr cs (code segment) wskazuje na segment zwierający bieżący ,wykonywany kod .CPU zawsze pobiera instrukcje z adresu podanego przez cs:ip. Domyślnie CPU oczekuje dostępu do większości zmiennych w segmencie danych .pewne zmienne i inne operacje występują w segmencie stosu .Kiedy uzyskujemy dostęp do danych w tych wyspecyfikowanych obszarach, żadna wartość segmentu nie jest konieczna. Przy dostępie do danych w jednym z tych ekstra segmentów (es ,fs lub gs),tylko jeden bajt jest potrzebny do wybrania właściwego rejestru segmentowego. Tylko kilka instrukcji sterujących przepływem pozwala nam wyspecyfikować pełny 32 bitowy adres segmentowy.

Teraz może się to wydawać ograniczeniem .W końcu tylko z czterema rejestrami segmentowymi w 8086 możemy zaadresować maksimum 256 kilobajtów (64K na segment),a nie cały obiecany megabajt. Jednakże możemy zmieniać rejestry segmentowe pod kontrolą programu, więc jest możliwe adresować każdy bajt poprzez zmianę wartości w rejestrze segmentowym.

Oczywiście, zabierze parę instrukcji zamiana wartości jednego z rejestrów segmentowych 80x86.Te instrukcje zużywają pamięć i zabierają czas na wykonanie .Tak więc zachowanie dwóch bajtów na dostęp do pamięci nie opłaca się, jeśli uzyskujemy dostęp do danych w różnych segmentach cały czas. Na szczęście, większość kolejnych dostępów do pamięci występuje w tym samym segmencie. W związku z tym, ładowanie rejestrów segmentowych nie jest czymś co robimy bardzo często.

4.6 TYBY ADRESOWANIA 80x86

Tak jak opisane procesory x86 w poprzednim rozdziale, procesory 80x86 pozwalają nam uzyskać dostęp do pamięci na wiele różnych sposobów. Tryby adresowania pamięci 80x86 dostarczają elastycznego dostępu do pamięci ,pozwalając nam na łatwy dostęp do zmiennych, tablic ,rekordów, wskaźników i innych złożonych typów danych. Biegłe opanowanie trybów adresowania 80x86 jest pierwszym krokiem do biegłego opanowania języka asemblera 80x86.

Kiedy Intel projektował oryginalny procesor 8086,wyposażył go w elastyczny, chociaż ograniczony ,zbiór trybów adresowania pamięci .Intel dodał kilka nowych trybów adresowania, kiedy wprowadzono mikroprocesor 80386.Zauważ,że 80386 zachował wszystkie tryby z poprzednich procesorów; nowe tryby zostały dodane jako dodatki. Kiedy musimy napisać kod który pracuje na procesorze 80286 lub wcześniejszych, nie będziemy mogli wykorzystać tych nowych trybów .Jednakże jeśli zamierzamy uruchomić nasz kod na procesorze 80386sx lub wyższych możemy użyć tych nowych trybów .Ponieważ wielu programistów pisze jeszcze programy które pracują na 80286 lub wcześniejszych maszynach ,jest ważne oddzielenie omawiania tych dwóch zbiorów trybów adresowania, aby uniknąć pomylenia ich.

4.6.1 TRYB ADRESOWANIA REJESTRÓW

Większość instrukcji 8086 może działać na zbiorze rejestrów ogólnego przeznaczenia 8086.Poprzez wyspecyfikowanie nazwy rejestru jako operandu instrukcji, możemy uzyskać dostęp do zawartości tego rejestru .Rozważmy instrukcję mov (move) 8086:

```
mov         przeznaczenie, źródło
```

Instrukcja ta kopiuje dane z operandu źródłowego do operandu przeznaczenia. Ośmio i szesnasto bitowe rejestry są z pewnością ważnymi operandami dla tej instrukcji .Ograniczeniem jest tylko to,że oba operandy muszą być tego samego rozmiaru .popatrzmy na kilka rzeczywistych instrukcji mov 8086:

```
mov         ax, bx          ;kopiuje wartość z bx do ax
mov         dl, al.         ;kopiuje wartość z al. do dl
mov         si, dx          ;kopiuje wartość z dx do si
mov         sp, bp          ;kopiuje wartość z bp do sp
mov         dh,cl           ;kopiuje wartość z cl do dh
mov         ax,ax           ;tak, to jest dozwolone!
```

Pamiętajmy ,że rejestry są najlepszym miejscem do trzymania często używanych zmiennych .Jak zobaczymy trochę później ,instrukcje używające rejestrów są krótsze i szybsze niż te które uzyskują dostęp do pamięci.. W całym tym rozdziale będziemy widzieć skrócone nazwy operandów reg i r/p. (rejestr/pamięć) używane wszędzie gdzie będziemy używali rejestrów ogólnego przeznaczenia.

Oprócz rejestrów ogólnego przeznaczenia, wiele instrukcji 8086 (wliczając w to instrukcję mov) pozwala nam wyspecyfikować jeden z rejestrów segmentowych jako operand. Są dwa ograniczenia przy używaniu rejestrów segmentowych z instrukcja mov. Po pierwsze ,nie możemy wyspecyfikować cs jako operandu przeznaczenia, po drugie tylko jeden z operandów może być rejestrem segmentowym. Nie możemy przenosić danych z jednego rejestru segmentowego do innego w pojedynczej instrukcji mov .Kopiowanie wartości z cs do ds. musi używać sekwencji instrukcji podobnej do tej:

```
mov         ax, cs
mov         ds., ax
```

Nie powinniśmy nigdy używać rejestrów segmentowych jako rejestru danych do przetrzymywania przypadkowych wartości. powinny one zwierać tylko adresy segmentów. Ale więcej o tym później .W całym tekście będziemy używać skróconych nazw operandów sreg ,wtedy kiedy rejestr segmentowy będzie przeznaczony (wymagany) jako operand.

4.6.2 TRYBY ADRESOWANIA PAMIĘCI 8086

8086 dostarcza 17 różnych sposobów dostępu do pamięci .Może to wydawać się to wydawać sporo z początku ale na szczęście większość trybów adresowania jest prostymi wariantami tak, że są one łatwe do nauczenia. A powinniśmy się ich nauczyć! Kluczem do dobrego programowania w asemblerze jest właściwe użycie trybów adresowania pamięci

Tryby adresowania dostarczone przez rodzinę 8086 obejmują „tylko –przemieszczenie”, bazowy, bazowy z przemieszczeniem, bazowy indeksowany i bazowy indeksowany z przemieszczeniem. Odmiany tych pięciu form dostarczają 17 różnych trybów adresowania w 8086.Zobaczmy ,z 17 do 5.Nie jest wcale tak źle!

4.6.2.1 TRYB ADRESOWANIA ‘TYLKO PRZEMIESZCZENIE ”

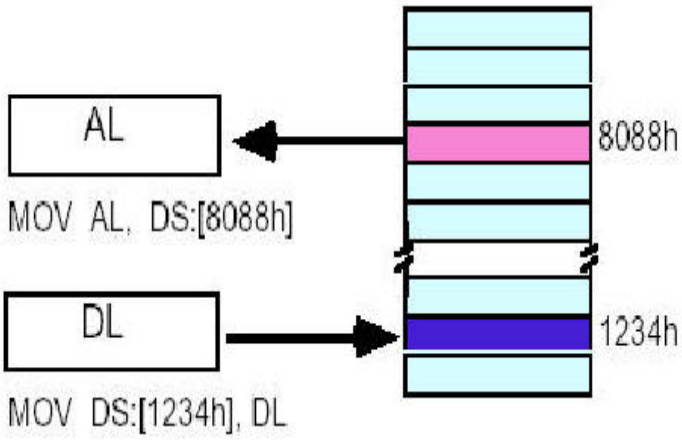
Najbardziej powszechnym trybem adresowania i jednym z łatwiejszych do zrozumienia, jest tryb adresowania „tylko przesunięcie” lub „bezpośredni”. Tryb adresowania „tylko przemieszczenie” składa się z 16 bitowej stałej która wyszczególnia adres lokacji docelowej .Instrukcja mov al .,ds:[8088h] ładuje do rejestru al. kopię bajtu spod komórki pamięci 8088h

Składnia MASM dla Trybów Adresowania Pamięci

Assembler z Microsoftu używa kilku różnych odmian na oznaczenie indeksowego, bazowo indeksowego i bazowo indeksowanego z przemieszczeniem trybów adresowania. Zobaczymy, że wszystkich tych form będziemy używać zamiennie w tym tekście. Następująca lista kilku możliwych kombinacji które są poprawne dla różnych trybów adresowania 80x86:

disp[bx],[bx] [disp],[bx+disp],[disp] [bx], i [disp+bx]
[bx] [si], [bx+si],[si] [bx], i [si+bx]
disp [bx] [si],disp[bx+si],[disp+bx+si],[disp+bx] [si],disp [si] [bx], [disp+si]
[bx],[disp+si+bx],[si+disp+bx],[bx+disp+si], itp.

MASM traktuje symbol „[ ]” jako operator „+”.Ten operator jest zamienny podobnie jak operator „+”.Oczywiście, to omówienie stosuje się we wszystkich trybach adresowania 8086,nie dotyczy to BX i SI .Możemy zastąpić każdy poprawny rejestr w powyższych trybach adresowania



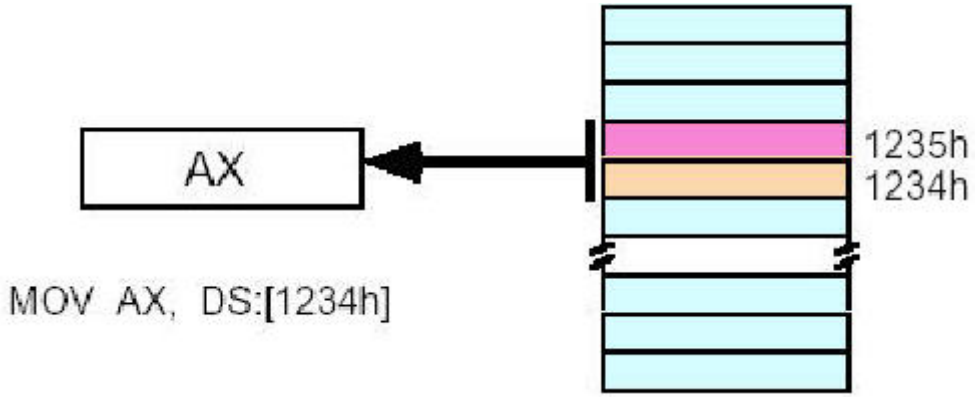
Rysunek 4.8: Tryb adresowania „Tylko przemieszczenie”(Bezpośredni)

Podobnie,instrukcja mov ds.:[1234h],dl zapamiętuje wartość rejestru dl w komórce pamięci 1234h (zobacz rysunek 4.8)

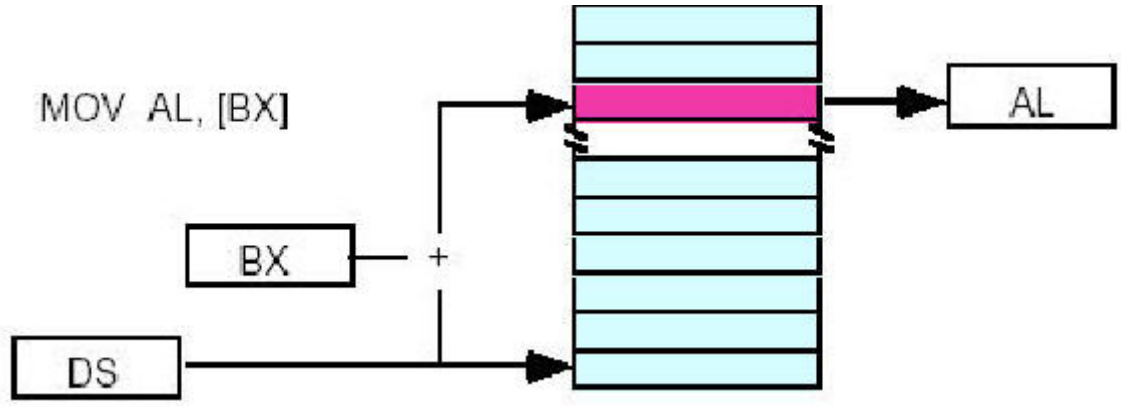
Tryb adresowania „tylko przemieszczenie” jest doskonały dla uzyskania dostępu do prostych zmiennych .Oczywiście, będziemy woleli używać nazwy takiej jak „I” lub „J” zamiast „DS.:[1234h]” lub „DS.:[8088h]”Cóż ,wkrótce zobaczymy ,że jest możliwe zrobić dokładnie tak.

Intel nazwał ten tryb trybem adresowania „tylko przemieszczenie” ponieważ 16 bitowa stała (przemieszczenie) występuje jako opcod instrukcji mov w pamięci .W tym względzie jest to całkiem podobne do bez[pośredniego trybu adresowania w procesorach x86 (zobacz poprzedni rozdział).Jest jednak kilka drobnych różnic .Przede wszystkim, przemieszczenie jest dokładnie tym – odległością od innego punktu .W x86,adres bezpośredni może być potraktowany jako przemieszczenie od adresu zero .W procesorach 80x86.to przesunięcie jest offsetem od początku segmentu (w tym przypadku segmentu danych)Nie martwmy się jeśli nie ma to dużego sensu w tej chwili. Dostaniemy możliwość do studiowania segmentów, trochę później w tym segmencie .Na razie możemy myśleć o trybie adresowania „tylko przemieszczeniu” jako bezpośrednim trybie adresowania. Przykłady w tym rozdziale będą uzyskiwać dostęp do bajtu pamięci .Nie zapomnij jednak ,że możemy również uzyskać dostęp do słowa w procesorach 8086 (zobacz rysunek 4.9)

Domyślnie ,wszystkie wartości „tylko przemieszczenia” dostarczają offsety do segmentu danych .Jeśli chcemy dostarczyć offset do innego segmentu, musimy użyć przedrostka przesłonięcia segmentu, przed naszym adresem .Na przykład ,aby uzyskać dostęp do komórki 1234h w ekstra segmencie(es) użyjemy instrukcji mov w postaci mov ax,es:[1234h].Podobnie ,aby uzyskać dostęp do komórki w segmencie kodu użyjemy instrukcji mov ax,cs:[1234h].Przedrostek ds. nie jest przesłonięciem segmentu. CPU używa rejestru segmentu danych domyślnie. Te określone przykłady wymagają ds: ponieważ jest to składniowe ograniczenie MASMa



Rysunek 4.9 Dostęp do słowa



Rysunek 4.10 Tryb adresowania [BX]

4.6.2.2 TRYB ADRESOWANIA POŚREDNIEGO PRZEZ REJESTR

CPU 80x86 pozwala uzyskać dostęp do pamięci pośrednio przez rejestr używając trybu adresowania pośredniego przez rejestr .Są cztery formy tego trybu adresowania w 8086,najlepiej dowieść na następujących instrukcjach:

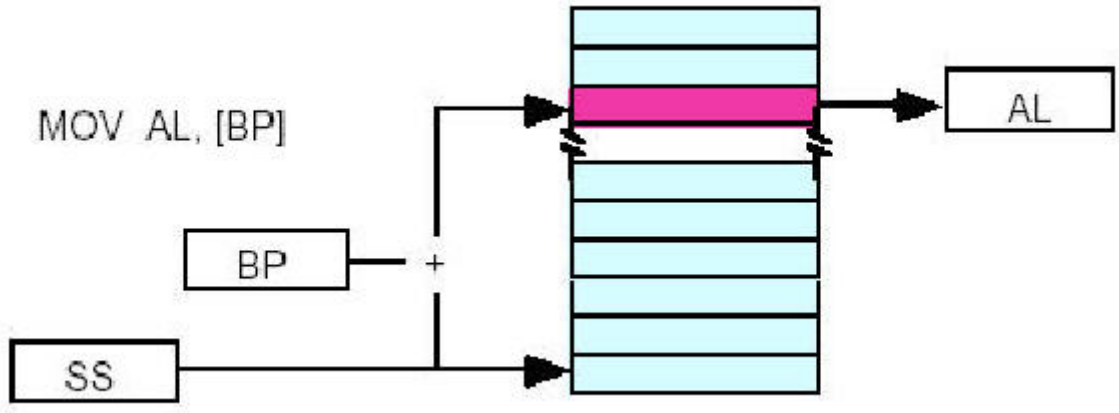
```
mov     al.,[bx]
mov     al.,[bp]
mov     al.,[si]
mov     al.,[di]
```

Możemy użyć przedrostka przesłonięcia segmentu, jeśli mamy życzenie uzyskać dostęp do danych w różnych segmentach .Następujące instrukcje demonstrują użycie przesłonięcia:

```
mov     al.,cs:[bx]
mov     al.,ds:[bp]
mov     al.,ss:[si]
mov     al.,es:[di]
```

Intel odnosi się do [bx] i [bp] jako trybu adresowania bazowego a bx i bp jako rejestrów bazowych(faktycznie bp oznacza wskaźnik bazowy).Intel odnosi się do trybu adresowania [si] i [di] jako trybu adresowania indeksowego (si oznacza indeks źródłowy ,di oznacza indeks przeznaczenia).jednak te tryby adresowania są funkcjonalnie równoważne.]

Zauważ: tryb adresowania [si] i [di] pracują dokładnie w ten sam sposób ,jak si i di dla bx powyżej.



Rysunek 4.11 Tryb adresowania [BP]

4.6.2.3 TRYB ADRESOWANIA INDEKSOWY

Tryb adresowania indeksowy używa następującej składni:

```
mov     al., disp[bx]
mov     al.,disp[bp]
mov     al.,disp[si]
mov     al.,disp[di]
```

Jeśli bx zawiera 1000h,wtedy instrukcja mov cl,20h[bx] załaduje cl z komórki pamięci ds.:1020h.Podobnie,jeśli bp zawiera 2020h, mov dh,1000h[bp] załaduje dh z komórki pamięci ss:3020.

Offset wytworzony w tym trybie adresowym to suma stałej i wyspecyfikowanego rejestru. Tryby adresowania wymagające bx,si i di wszystkie używają segmentu danych, tryb adresowania disp[bp] używa domyślnie segmentu stosu. Przy korzystaniu z trybu adresowania pośredniego przez rejestr, możemy użyć przedrostków przesłonięcia segmentu dla wyspecyfikowania innego segmentu:

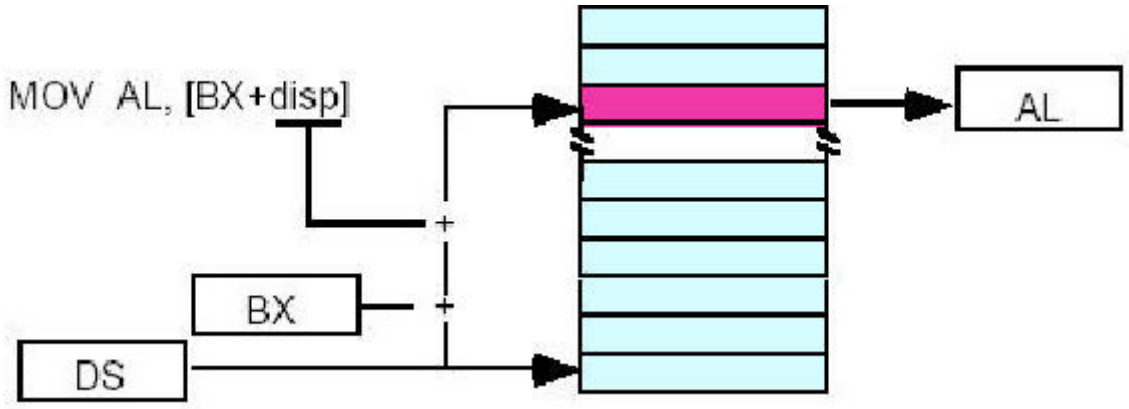
```
mov     al.,ss:disp[bx]
mov     al.,es:disp[bp]
mov     al.,cs:disp[si]
mov     al.,ss:disp[di]
```

Możemy zastąpić si lub di z rysunku 4.12 trybem adresowania [si+disp] i [di+disp].Zauważ, że Intel odnosi się do tych trybów adresowania jako adresowania bazowego i adresowania indeksowego. Literatura Intela nie rozróżnia pomiędzy tymi trybami z lub bez stałej .Jeśli popatrzymy na to jako pracę sprzętową, jest to sensowna definicja. Z programistycznego punktu widzenia te tryby adresowania są całkowicie użyteczne

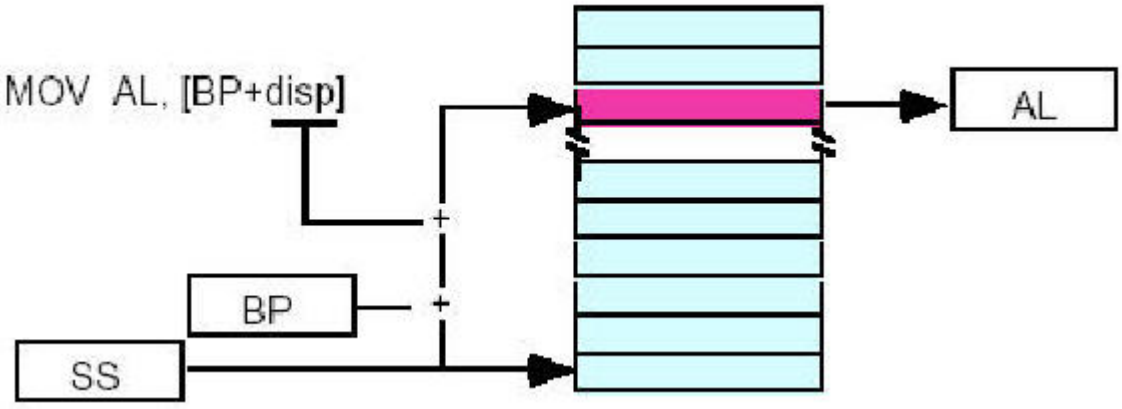
---

ADRESOWANIE BAZOWE I INDEKSOWE

W rzeczywistości jest subtelna różnica między bazowym a indeksowym adresowaniem. Oba tryby adresowania składają się z przesunięcia dodanego razem z rejestrem. Główną różnica między nimi jest względny rozmiar wartości przesunięcia i rejestru .W trybie adresowania indeksowego stała dostarcza adresu wyspecyfikowanej struktury danych a rejestr dostarcza offsetu względem tego adresu. W trybie adresowania bazowego, rejestr zawiera adres struktury danych a stała dostarcza przemieszczenia liczonego od tego punktu. ponieważ dodawanie jest przemienne, te dwie prezentacje są zasadniczo równoważne. ponieważ Intel obsługuje jeden i dwa bajty przemieszczenia (zobacz ”Instrukcja MOV 80x86”) większy sens będzie miało nazywanie ich trybem adresowania bazowego.



Rysunek 4.12 Tryb Adresowania [BX+disp]



Rysunek 4.13 Tryb Adresowania

dla innych rzeczy .Dlatego ten tekst używa różnych terminów do ich opisu. Niestety jest bardzo mała jednomyślność w używaniu tych terminów w świecie 80x86.

4.6.2.4 TRYB ADRESOWANIA BAZOWY INDEKSOWANY

Tryb adresowania bazowy indeksowany jest po prostu kombinacją trybu adresowania pośredniego przez rejestr. Te tryby adresowania tworzy się poprzez dodanie offsetu razem z rejestrem bazowym (bx lub bp) i rejestrem indeksowym (si lub di).Dopuszczalne formy tego trybu adresowania to:

```
mov     al., [bx] [si]
mov     al., [bx] [di]
mov     al., [bp] [si]
mov     al., [bp] [di]
```

Przypuśćmy ,że bx zawiera 1000h a si 880h.Wtedy instrukcja

```
mov     al.,[bx] [si]
```

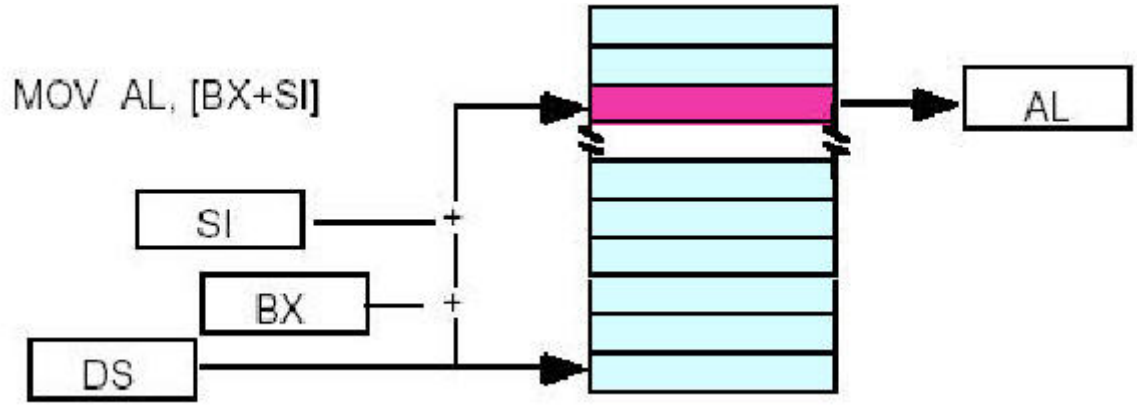
będzie ładowała al. z komórki DS:1880h.Podobnie jeśli bp zawiera 1598h a di 1004,
mov ax,[bp+di] załaduje 16 bitów w ax z komórki SS:259C i SS:259D.

Tryb adresowania który nie wymaga bp używa domyślnie segmentu danych .Jeśli używamy bp jako operandu, domyślnie używamy segmentu stosu.

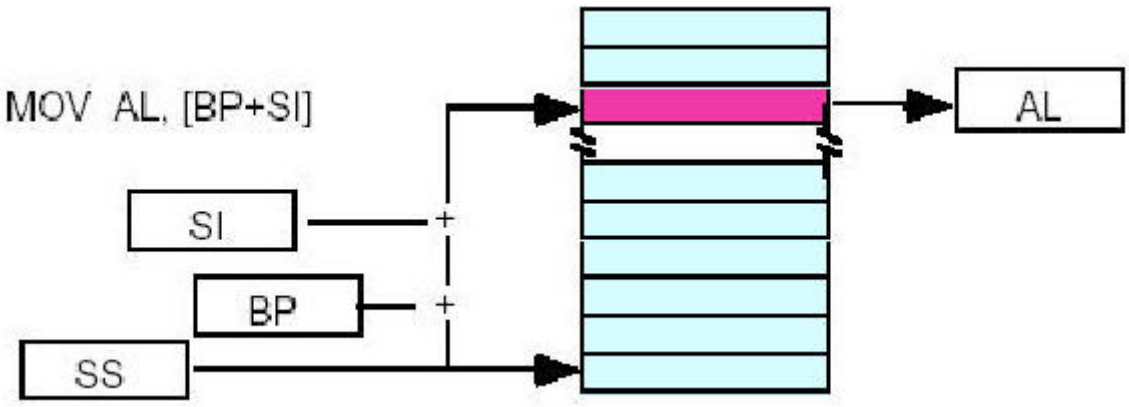
Podstawimy di w rysunku 4.12 uzyskamy tryb adresowania [bx+di].Podstawiamy di w rysunku 4.13 dla trybu adresowania [bp+di].

4.6.2.5 TRYB ADRESOWANIA BAZOWY INDEKSOWANY Z PRZEMIESZCZENIEM

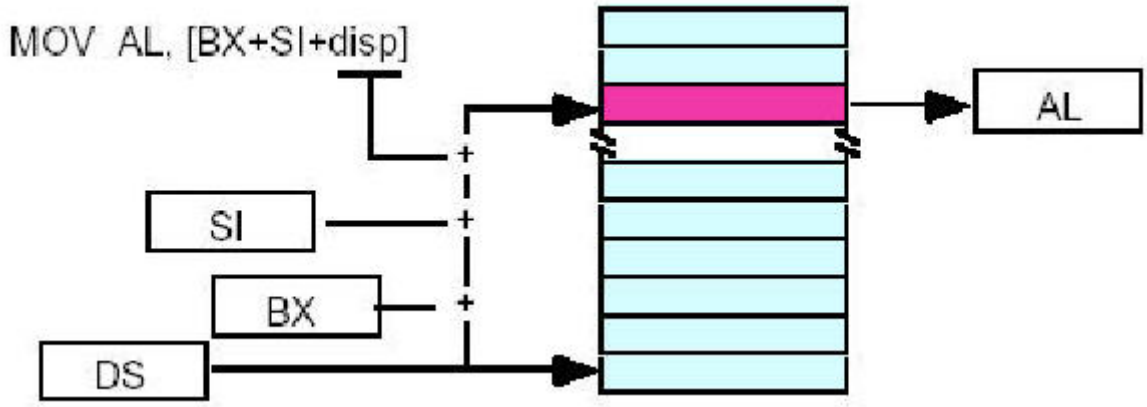
Ten tryb adresowania jest drobną modyfikacją trybu adresowania bazowego /indeksowego z dodaniem ośmio- lub szesnastobitowej stałej. Poniżej są przykłady tego trybu adresowania (zobacz rysunek 4.14 i rysunek 4.15)



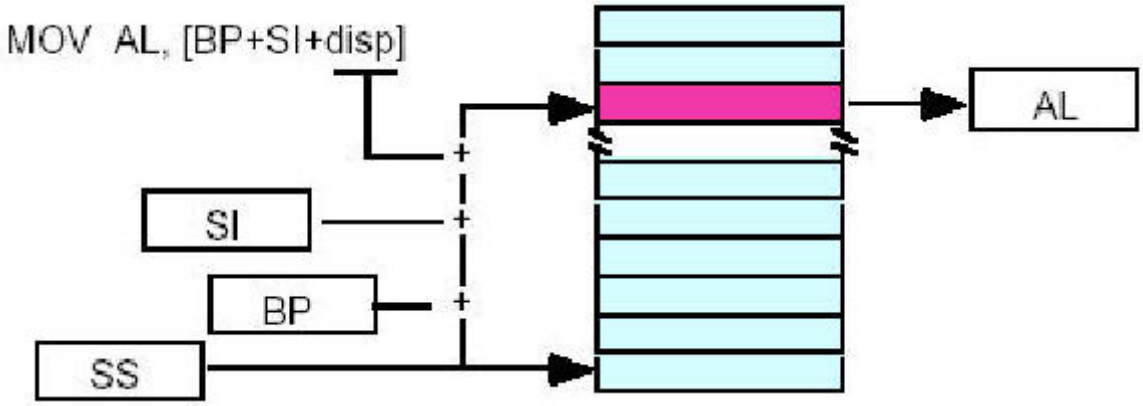
Rysunek 4.14 Tryb adresowania [BX+SI]



Rysunek 4.15 Tryb adresowania [BP+SI]



Rysunek 4.16 Tryb adresowania [BX+SI+disp]



Rysunek 4.17 Tryb adresowania [BP+SI+disp]

```
mov     al.,disp[bx] [si]
mov     al.,disp[bx+di]
mov     al.,[bp+si+disp]
mov     al.,[bp] [di] [disp]
```

Możemy zamienić di w rysunku 4.16 i stworzymy tryb adresowania [bx+di+disp].Możemy zamienić di w rysunku 4.17 i stworzymy tryb adresowania [bp=di+disp].

Rysunek 4.18 Tablica generująca poprawne tryby adresowania 8086.

Przypuśćmy, że bp zawiera 1000h,bx 2000h,si 120h a di 5.Wtedy mov al.,10h[bx+si] ładuje al spod adresu DS.:2130; mov ch,125h[bp+di] ładuje ch z komórki SS:112A; a mov bx,cs:2[bx][di] ładuje bx z komórki CS: 2007.

4.6.2.6 ŁATWY SPOSÓB NA ZAPAMIĘTANIA TRYBÓW ADRESOWANIA PAMIĘCI

Ogólna liczba poprawnych trybów adresowania w 8086 wynosi 17: disp,[bx],[bp],[si],[di],disp[bx],disp[bp],disp[si],disp[di],[bx][si],[bx][di],[bp][si],[bp][di],disp[bx][si],disp[bx][di],disp[bp][di] i disp[bp][si].Możemy wprowadzić do pamięci wszystkie te formy żeby poznać ,który jest poprawny (i przez pominięcie, które są nieprawidłowe).Jednakże jest łatwiejszy sposób poza wprowadzaniem do pamięci tych 17 form. Rozpatrzmy tablicę z rysunku 4.18.

Jeśli wybierzemy pozycję zero lub jeden z każdej kolumny i zakończymy przynajmniej jedną pozycją, otrzymamy prawidłowy tryb adresowania pamięci 8086.Kilka przykładów:

- Wybierzmy disp z kolumny jeden, nic z kolumny drugiej,[di] z kolumny 3,otrzymamy disp [di]
- Wybierzmy disp,[bx] i [di].Mamy disp[bx][di]
- Pomińmy kolumny jeden i dwa, wybieramy [si].Mamy [si]
- Pomińmy kolumnę jeden, wybieramy [bx],potem wybieramy [di].mamy [bx][di]

Podobnie, jeśli mamy tryb adresowania, którego nie możemy zbudować z tej tablicy, wtedy nie jest to poprawne. Na przykład ,disp[dx][si] jest nielegalne, ponieważ nie możemy otrzymać [dx] z żadnej z kolumny powyżej.

4.6.2.7 KILKA KOŃCOWYCH UWAG O TRYBACH ADRESOWANIA 8086

Adres efektywny jest końcowym offsetem stworzonym przez obliczenia trybu adresowego. Na przykład ,jeśli bx zawiera 10h,adres efektywny dla 10h[bx] to 20h.Zauważmy,że termin adres efektywny występuje w prawie każdym omówieniu trybów adresowania 8086.Jest nawet specjalna instrukcja ładuj adres efektywny (lea) który oblicza adres efektywny.

Nie wszystkie tryby adresowania są tworzone równo! Różne tryby adresowania mogą zabierać różną ilość czasu do obliczenia adresu efektywnego. Dokładne różnice zmieniają się z procesora na procesor. Generalnie, im bardziej złożony tryb adresowania tym dłużej trwa obliczanie adresu efektywnego. Złożoność trybu adresowania jest bezpośrednio powiązana z liczbą elementów w trybie adresowania .Na przykład, disp[bx][si] jest bardziej złożona niż [bx].Zobacz zbiór instrukcji w dodatkach do informacji odnośnie czasu cyklu różnych trybów adresowania na różnych procesorach 80x86.

Pole przemieszczenia we wszystkich trybach adresowania z wyjątkiem „tylko przemieszczenie" może być ośmiobitową stałą ze znakiem lub 16 bitową stałą ze znakiem .Jeśli offset jest z zakresu -128...+127 instrukcje będą krótsze (a zatem szybsze) niż instrukcje z przemieszczeniem poza tym zakresem. Rozmiar wartości w rejestrze nie wpływa na czas wykonania lub rozmiar .Więc jeśli możemy wstawić dużą liczbę do rejestru(ów) i użyć małego przemieszczenia ,jest bardziej pożądane duża stała i mała wartość w rejestrze.

Jeśli adres efektywny po obliczeniach tworzy wartość większą niż 0FFFFh,CPU ignoruje przepełnienie a wynik przechodzi cyklicznie z powrotem do zera. Na przykład, jeśli bx zawiera 10h,wtedy instrukcja mov al.,0FFFFh[bx] załaduje rejestr al. z komórki ds.:0Fh nie z komórki 1000Fh.

W tym omówieniu widzieliśmy jak działają te tryby adresowe. Poprzednie omówienie nie wyjaśniło dla czego ich używamy. Przyjdzie to trochę później. tak długo jak wiemy jak każdy tryb adresowania wykonuje obliczanie swojego adresu efektywnego, będzie dobrze.

4.6.3 TRYB ADRESOWANIA REJESTRÓW 80386

Procesory 80386 (i późniejsze) posiadają 32 bitowe rejestry .Wszystkie osiem rejestrów ogólnego przeznaczenia ma swoje 32 bitowe odpowiedniki .Są to eax,ebx,ecx,edx,esi,edi,ebp i esp. Jeśli używamy procesora 80386 lub późniejszego, możemy używać tych rejestrów jako operandów dla kilku instrukcji 80386.

4.6.4 TRYBY ADRESOWANIA PAMIĘCI

Procesor 80386 uogólnia tryby adresowania pamięci .Podczas gdy 8086 pozwalał nam tylko na użycie bx lub bp jako rejestrów bazowych a si lub di jako rejestrów indeksowych,80386 pozwala nam używać prawie każdego 32 bitowego rejestru ogólnego przeznaczenia jako rejestru bazowego lub indeksowego. Co więcej,80386 wprowadza nowy tryb adresowania indeksowego ze skalowaniem, który upraszcza uzyskanie dostępu do elementów tablic .Poza zwiększeniem do 32 bitów, nowy tryb adresowania w 80386 jest prawdopodobnie największym ulepszeniem chipu w stosunku do wcześniejszych procesorów.

4.6.4.1 TRYB ADRESOWANIA POŚREDNIEGO PRZEZ REJESTR

W 80386 możemy wyspecyfikować każdy 32 bitowy rejestr ogólnego przeznaczenia kiedy używamy trybu adresowania pośredniego przez rejestr. [eax],[ebx],[ecx],[edx],[esi] i [edi] zapewniają domyślnie offset w segmencie danych. tryby adresowania [ebp] i [esp] używają domyślnie segmentu stosu.

Zauważ ,że podczas pracy w 16 bitowym trybie rzeczywistym w 80386,offset w tych 32 bitowych rejestrach musi być z zakresu 0 ....0FFFFh.Nie możemy użyć wartości większych niż ta do której uzyskujemy dostęp więcej niż 64K w segmencie. Zauważ również musimy używać 32 bitowych nazw rejestrów .Nie możemy użyć nazw 16 bitowych .Następujące instrukcje demonstrują wszystkie poprawne formy:

```
mov     al.,[eax]
mov     al.,[ebx]
mov     al.,[ecx]
mov     al.,[edx]
mov     al.,[esi]
mov     al.,[edi]
mov     al.,[ebp] ; używa SS domyślnie
```

4.6.4.2 TRYBY ADRESOWANIA INDEKSOWY,BAZOWY INDEKSOWANY,BAZOWY INDEKSWANY Z PRZEMIESZCZENIEM 80386

Tryb adresowania indeksowego (pośrednio przez rejestr plus przemieszczenie) pozwala nam połączyć 32 bitowy rejestr ze stałą. Tryb adresowania bazowego/ indeksowego pozwala nam łączyć dwa 32 bitowe rejestry .W końcu tryb adresowania bazowy indeksowany z przemieszczeniem pozwala nam łączyć stałą i dwa rejestry do sformowania adresu efektywnego .Zapamiętajmy, że offset stworzony przez obliczenie adresu efektywnego musi być długi na szesnaście bitów jeśli działa w trybie rzeczywistym.

W 80386 termin rejestr bazowy i rejestr indeksowy właściwie mają takie samo znaczenie .Kiedy łączymy dwa 32 bitowe rejestry w trybie adresowania ,pierwszy rejestr jest rejestrem bazowym a drugi rejestrem indeksowym .Jest to prawda bez względu na nazwy rejestrów. Zauważ ,że 80386 pozwala nam użyć tego samego rejestru jako obu, rejestru bazowego indeksowego, które są właściwie użyteczne czasami .Następujące instrukcje dostarczają reprezentatywnych przykładów trybów adresowania bazowego i indeksowego w różnymi wariantami składniowymi:

```
mov     al.,disp[eax]
mov     al.,[ebx+disp]
mov     al.,[ecx] [disp]
mov     al.,disp[edx]
mov     al.,disp[esi]
mov     al.,disp[edi]
mov     al.,disp[ebp]
mov     al.,disp[esp]
```

Następujące instrukcje wszystkie używają trybu adresowania bazowego +indeksowego .Pierwszy rejestr w drugim operandzie jest rejestrem bazowym, drugim jest rejestr indeksowy.. Jeśli rejestr bazowy to esp lub ebp adres efektywny używa segmentu stosu. W przeciwnym razie ,adres efektywny używa segmentu danych. Zauważ, że wybór rejestru indeksowego nie wpływa na wybór segmentu domyślnego.

```
mov     al.,[eax][ebx]
mov     al.,[ebx+edx]
mov     al.,[ecx][edx]
mov     al.,[edx][ebp]
mov     al.,[esi][edi]
mov     al.,[edi][esi]
mov     al.,[ebp+ebx]
mov     al.,[esp][ecx]
```

Naturalnie, możemy dodać przemieszczenie do powyższych trybów adresowania, stworzymy tryb adresowania bazowy indeksowany z przemieszczeniem .Następujące instrukcje pokazują reprezentatywne przykłady możliwych trybów adresowania:

```
mov     al., disp[eax][ebx]
mov     al., disp[ebx+ebx]
mov     al.,[ecx+edx+disp]
mov     al., disp[edx+ebp]
mov     al.,[esi][edi] [disp]
mov     al.,[edi][disp][esi]
mov     al., disp[ebp+ebx]
mov     al.,[esp+ecx][disp]
```

Jest jedno ograniczenie w 80386 przy ustalaniu rejestru indeksowego. Nie możemy użyć rejestru esp jako rejestru indeksowego. Jest OK., jeśli użyjemy esp jako rejestru bazowego, ale nie jako rejestru indeksowego.

4.6.4.3TRYB ADRESOWANIA INDEKSWOEGO ZE SKALOWANIEM

Tryby adresowania indeksowy, bazowy /indeksowy i bazowo indeksowy z przemieszczeniem są specjalnymi przypadkami trybu adresowania indeksowego ze skalowaniem. Te tryby adresowania są szczególnie użyteczne przy dostępie do elementów tablicy, chociaż nie są one ograniczone tylko do tego celu. Te tryby pozwolą nam pomnożyć rejestr indeksowy w trybie adresowania przez jeden, dwa, cztery lub osiem. Ogólna składnia tego trybu adresowania

```
disp[index*n]
[base][index*n]
```

lub

```
disp[base][index*n]
```

gdzie „baza" i „indeks" reprezentują każdy z rejestrów ogólnego przeznaczenia 80386 a n jest to wartość jeden ,dwa, cztery lub osiem.

80386 oblicza adres efektywny przez dodanie disp ,bazy i indeks*n razem. Na przykład ,jeśli ebx zawiera 1000h a esi 4 wtedy

```
mov     al.,8[ebx][esi*4]       ;ładuje AL. z komórki 1018h
mov     al.,1000h[ebx][ebx*2]   ;ładuje AL. z komórki 4000h
mov     al.,1000h[esi*8]        ;ładuje AL. z komórki 1020h
```

Zauważ ,że 80386 rozszerza tryby adresowania indeksowy, bazowy indeksowany, bazowo indeksowany z przemieszczeniem naprawdę jako specjalne przypadki trybu adresowania indeksowego ze skalowaniem z n równym jeden. To znaczy, następujące pary instrukcji są absolutnie identyczne dla 80386:

```
mov     al.,2[ebx][esi*1]       mov     al.,2[ebx][esi]
mov     al.,[ebx][esi*1]        mov     al.,[ebx][esi]
mov     al.,2[esi*1]            mov     al.,2[esi]
```

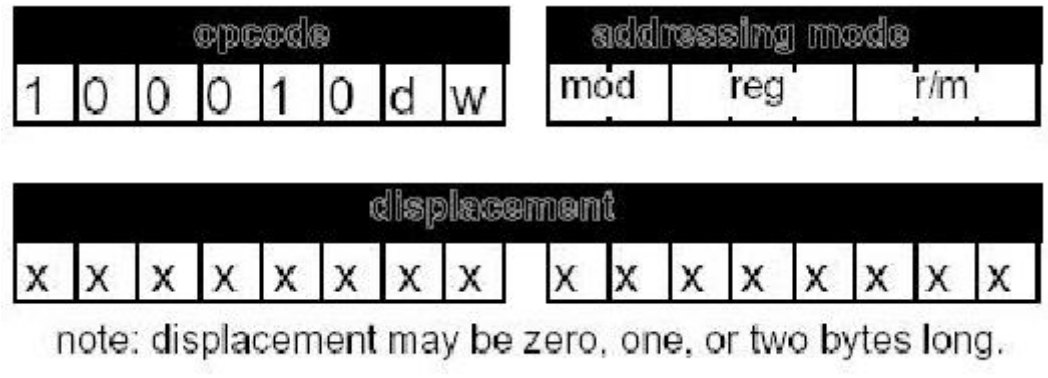
Oczywiście .MASM pozwala na mnóstwo różnych wariantów tych trybów adresowania .Poniżej mamy kilka możliwych przykładów:

Disp [bx][si*2],[bx+disp][si*2],[bx+si*2+disp],[si*2+bx][disp], disp[si*2][bx],[si*2+disp][bx],[disp+bx][si*2]

4.6.4.4 KILKA KOŃCOWYCH UWAG O TRYBACH ADRESOWANIA W 80386

Ponieważ tryby adresowania 80386 są bardziej ortogonalne, mogą one dużo łatwiej wprowadzać do pamięci niż tryby adresowania 8086.Dla programistów pracujących na procesorach 80386 istnieje zawsze pokusa do pomijania trybów adresowania 8086 i używania wyłącznie zbiór 80386.Jednakże,jak zobaczymy w następnej sekcji ,tryby adresowania 8086 naprawdę są bardziej wydajne niż porównywalne tryby adresowania 80386.Zatem ważne jest aby poznać wszystkie tryby adresowania i wybrać tryb odpowiedni dla danego problemu.

Kiedy używamy trybu adresowania bazowego/indeksowego i bazowego indeksowanego z przemieszczeniem w 80386 bez operacji skalowania, pierwszy rejestr staje się w trybie adresowania rejestrem bazowym a rejestr drugi rejestrem indeksowym. Jest to ważny punkt ponieważ wybór domyślnego segmentu dokonuje się przez wybór rejestru bazowego. Jeśli rejestr bazowy to ebp lub esp,80386 domyślnie używa segmentu stosu. We wszystkich innych przypadkach 80386 uzyskuje dostęp domyślnie do segmentu danych, nawet jeśli rejestr indeksowy to ebp. Jeśli używamy operatora indeksu skalowania(„*n") w rejestrze ;ten rejestr jest zawsze rejestrem indeksowym bez względu na to gdzie pojawia się on w trybie adresowym:



Rysunek 4.19 Ogólna instrukcja MOV

```
[ebx][ebp]          ;używa domyślnie DS.
[ebp][ebx]          ;używa domyślnie SS
[ebp*1][ebx]        ;używa domyślnie DS.
```

```
[ebx][ebp*1]            ;używa  domyślnie DS.
[ebp][ebx*1]            ;używa domyślnie SS
[ebx*1][ebp]            ; używa domyślnie SS
es:[ebx][ebp*1]         ;używa ES
```

4.7 INSTRUKCJA MOV 80x86

Przykłady w tym rozdziale dość obszernie używają instrukcję (move) mov 80x86.Co więcej, instrukcja mov jest najpowszechniejszą instrukcją maszynową 80x86.Zatem,jest warte zachodu spędzić kilka chwil na omówieniu działania tej instrukcji.

Jako odpowiednik x86,instrukcja jest bardzo prosta. Przybiera formę:

```
mov         przez, źródło
```

Mov robi kopię Źródła i przechowuje tą wartość w Przez. instrukcja nie wpływa na oryginalną zawartość Źródła. Nadpisuje poprzednią wartość w Przez. Przeważnie ,operacje tej instrukcji można opisać wyrażeniem pascalowskim:

```
Przez := Źródło;
```

Ta instrukcja ma wiele ograniczeń .Dostaniemy pokaźną możliwość zajmowania się nimi przez cały czas studiowania języka asemblera 80x86.Zrozumienie dlaczego te ograniczenia istnieją, przypatrzymy się kodom maszynowym dla kilku różnych postaci tej instrukcji .Kodowanie dla instrukcji mov jest prawdopodobnie najbardziej złożoną w zbiorze instrukcji. Pomimo to ,bez studiowania kodu maszynowego tej instrukcji nie będziemy zdolni do jej docenienia. ani nie będziemy mieli pełnego zrozumienia jak pisać optymalne kody używając tej instrukcji. Zobaczymy dlaczego pracowaliśmy procesorami x86 w poprzednim rozdziale zamiast używać rzeczywistych instrukcji 80x86.

Jest kilka wersji instrukcji mov. Mnemonik mov opisuje dwanaście różnych instrukcji w 80386.Najbardziej powszechne użycie instrukcji mov ma następujące binarne kodowanie pokazane na rysunku 4.19. Opcodem jest pierwsze osiem bitów instrukcji. Bity zero i jeden definiują szerokość instrukcji (8,16 lub 32 bity) i kierunek przeniesienia. Kiedy omawiamy specyficzne instrukcje w tym tekście zawsze wypełniamy wartości d i w dla siebie. Pojawiają się tu tylko dlatego,że prawie w każdym innym tekście na ten temat wymagane jest aby wypełnić te wartości.

Następujące opcody są adresowane bajtem czule nazywanym bajtem „mod-reg-r/m.” przez większość programistów. Ten bajt wybiera z 256 różnych możliwych kombinacji operandów pozwalających generować instrukcje mov. Ogólna instrukcja mov przybiera trzy różne formy w języku asemblera:

```
mov         reg, pamięć
mov         pamięć, reg
mov         reg,  reg
```

Zauważ, że co najmniej jeden z tych operandów jest rejestrem ogólnego przeznaczenia. Pole reg w bajcie mod/reg/rm specyfikuje ten rejestr.(lub jeden z rejestrów jeśli używamy trzech powyższych form).Bit d (direction –kierunku) w opcodzie decyduje czy przechowywanie danych będzie w rejestrze (d=1) lub pamięci (d=0).

Bity w polu reg pozwalają wyselekcjonować jeden z ośmiu różnych rejestrów.8086 wspiera 8 ośmiobitowych rejestrów i 8 szesnastobitowych rejestrów ogólnego przeznaczenia.80386 również wspiera osiem 32 bitowych rejestrów ogólnego przeznaczenia. CPU dekoduje znaczenie pole reg jak następuje:

reg	w=0	16 bit mode w=1	32 bit mode w=1
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

Tablica 23:kodowanie bitu REG

Rozróżniając 16 i 32 bitowe rejestry,80386 i późniejsze procesory używając specjalnych przedrostków bajtów opcodów przed instrukcjami używającymi 32 bitowych rejestrów. W przeciwnym razie, kodowanie instrukcji następuje w ten sam sposób dla obu typów instrukcji.
Pole r/m. ,wraz z polem mod, wybierają tryb adresowania. Pole mod jest kodowane jak następuje:

MOD	Meaning
00	The r/m field denotes a register indirect memory addressing mode or a base/indexed addressing mode (see the encodings for r/m) *unless* the r/m field contains 110. If MOD=00 and r/m=110 the mod and r/m fields denote displacement-only (direct) addressing.
01	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is an eight bit signed displacement following the mod/reg/rm byte.
10	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is a 16 bit signed displacement (in 16 bit mode) or a 32 bit signed displacement (in 32 bit mode) following the mod/reg/rm byte .
11	The r/m field denotes a register and uses the same encoding as the *reg* field

Tabela 24 Kodowanie MOD

Pole mod wybiera między przesunięciem rejestr-do-rejestru i przesunięcia rejestr-do/z-pamięci. Wybiera również rozmiar przemieszczenia(zero, jeden, dwa lub cztery bajty) który występuje przy trybie adresowania pamięci. Jeśli MOD=00,wtedy mamy jeden z trybów adresowania bez przemieszczenia (pośrednie przez rejestr lub bazowe / indeksowe).Zauważmy szczególny przypadek gdzie MOD=00 a r/m.=110.Będzie to odpowiadało trybowi adresowaniu [bp].8086 używa tego kodowania dla trybu adresowania „tylko-przemieszczenie". To oznacza, że nie jest prawdziwy tryb adresowania [bp] w 8086.

Aby zrozumieć dlaczego możemy używać trybu adresowania [bp] w naszych programach, popatrzmy na MOD=01 i MOD=10 w powyższej tabeli Te bity próbują uruchomić tryby adresowania disp[reg] i disp [reg][reg]."Więc co? Nie jest to samo co tryb adresowania [bp]"racja. Jednak rozważmy następujące instrukcje:

```
mov     al.,0[bx]
mov     ah,0[bp]
mov     0[si],al.
mov     0[di],ah
```

Te wyrażenia używając trybu adresowania indeksowego, wykonują takie same operacje jak ich odpowiedniki w trybie adresowania pośredniego przez rejestr (uzyskując to poprzez usunięcie przemieszczenia z powyższych instrukcji).Prawdziwa różnica między tymi dwoma formami jest to ,że tryb adresowania indeksowego jest długi na jeden bajt(jeśli MOD=01,długi na dwa bajty jeśli MOD=10) do utrzymania zerowego przemieszczenia. Ponieważ są one długie, te instrukcje mogą również pracować trochę wolniej.

Ta cech 8086 – dostarcza dwa lub więcej sposobów osiągnięcia tej samej rzeczy-pojawia się w całym zbiorze instrukcji. MASM generalnie wybiera najlepsze formy instrukcji automatycznie .Jeśli zapiszemy powyższy kod i zasemblujemy go używając MASM, wygeneruje tryb adresowania pośredniego przez rejestr dla wszystkich instrukcji za wyjątkiem mov ah,0[bp].Wyemituje tylko jednobajtowe przemieszczenie które jest krótsze i szybsze niż ta sam instrukcja z zerowym przemieszczeniem dwubajtowym .zauważmy ,że MASM nie wymaga żeby zapisać 0[bp],możemy zapisać [bp] a MASM automatycznie wstawi bajt zero przed [bp].

Jeśli MOD nie równa się 11b,pole r/m. koduje tryb adresowania pamięci jak następuje:

R/M	Addressing mode (Assuming MOD=00, 01, or 10)
000	[BX+SI] or DISP[BX][SI] (depends on MOD)
001	[BX+DI] or DISP[BX+DI] (depends on MOD)
010	[BP+SI] or DISP[BP+SI] (depends on MOD)
011	[BP+DI] or DISP[BP+DI] (depends on MOD)
100	[SI] or DISP[SI] (depends on MOD)
101	[DI] or DISP[DI] (depends on MOD)
110	Displacement-only or DISP[BP] (depends on MOD)
111	[BX] or DISP[BX] (depends on MOD)

Tablica 25 Kodowanie Pola R/M.

Nie zapomnijmy ,że tryby adresowania wymagają aby z bp używał segmentu stosu (ss) domyślnie. Wszystkie inne używają domyślnie segmentu danych (DS.).

Jeśli to omówienie namieszało nam w głowach ,nie widzieliśmy jeszcze gorszych rzeczy. Zapamiętajmy ,jest kilka trybów adresowania 8086.Przyjrzeliśmy się wszystkim trybom adresowania 80386.Prawdopodobnie zaczęliśmy rozumieć co znaczy, kiedy mówimy o złożonym zbiorze instrukcji .Jednak, ważnym pojęciem jest to,że możemy zbudować instrukcje 80x86 w ten sam sposób jak zbudowano instrukcje x86 w Rozdziale Trzecim – przez zbudowanie instrukcji bit po bicie .Po pełne szczegóły jak 80x86 koduje instrukcje zajrzymy do dodatków.

4.8 KILKA KOŃCOWYCH UWAG O INSTRUKCJI MOV

Jest kilka ważnych faktów o których powinniśmy pamiętać ,przy instrukcji mov. Przede wszystkim, nie ma przesunięcia z pamięci do pamięci .Z tego samego powodu, nowi użytkownicy języka asemblera mają ciężko przyswoić ten punkt. Podczas gdy jest para instrukcji które wykonują przesunięcie z pamięci do pamięci ,ładowanie rejestru a potem przechowywania tego rejestru prawie zawsze wydajnie. Innym ważnym faktem do zapamiętania o instrukcji mov, jest to,że jest wiele różnych instrukcji mov które osiągają te same rzeczy. Podobnie, jest kilka różnych trybów adresowania których możemy używać przy dostępie do tej samej komórki pamięci. Jeśli jesteśmy zainteresowani pisaniem możliwie najkrótszych i najszybszych programów w asemblerze ,musimy stale być świadomi różnic między odpowiednim instrukcjami.

W tym rozdziale zajmowaliśmy się głównie omawianiem ogólnej instrukcji mov, więc mogliśmy zobaczyć jak procesory 80x86 kodują tryby adresowania pamięci i rejestrów w instrukcji mov .Inne formy instrukcji mov pozwalają nam przenosić dane między 16 bitowymi rejestrami ogólnego przeznaczenia i rejestrami segmentowymi 80x86.Inne pozwalają nam załadować rejestry lub komórki pamięci stałymi. Te warianty instrukcji mov używają różnych opcodów .Po więcej szczegółów zajrzyjmy do kodowania instrukcji w Dodatku D

Jest kilka dodatkowych instrukcji mov w 80386 które pozwalają nam załadować rejestry specjalnego przeznaczenia 80386.W tym tekście ich nie rozpatrywaliśmy .Są również instrukcje łańcuchowe w 80x86 które wykonują operacje pamieć do pamięci. Takie instrukcje pojawią się w następnym rozdziale. To nie są dobrym substytutem dla instrukcji mov.

4.11 PODSUMOWANIE

Ten rozdział przedstawił organizację pamięci i strukturę danych 80x86.Nie jest to oczywiście kompletny kurs o strukturze danych, faktycznie ten temat pojawi się znowu później w Tomie Drugim. Ten rozdział omawia prymitywne i proste połączenie typów i danych i w jaki sposób deklarować i używać ich w naszych programach. Mnóstwo dodatkowych informacji na temat deklarowania i używania prostych typów danych pojawi się w „MASM :Dyrektywy i Pseudo-Opcody".

8088,8086,80188,80186 i 80286 wszystkie dzielą powszechny zbiór rejestrów których używają typowe programy. Ten zbiór rejestrów zawiera rejestry ogólnego przeznaczenia :ax,bx,cx,dx,si,di,bp i sp; rejestry segmentowe: cs, ds .,es i ss; i rejestry specjalnego przeznaczenia ip i flagi. Te rejestry są szerokie na szesnaście bitów ,Te procesory maja również 8 ośmiobitowych rejestrów: al .ah .bl .bh .cl .ch .dl i dh które nakładają się na rejestry ax,bx,cx i dx .Zobacz:

- 8086 Rejestry Ogólnego Przeznaczenia
- 8086 Rejestry Segmentowe
- 8086 Rejestry Specjalnego Przeznaczenia

W dodatku,80286 wspiera kilka rejestrów specjalnego przeznaczenia do zarządzania pamięcią, które są użyteczne w systemie operacyjnym i innych programów z poziomu systemu. Zobacz:

- Rejestry 80286

80386 i późniejsze procesory rozszerzają zbiór rejestrów ogólnego i specjalnego przeznaczenia do 32 bitów. Te procesory również dodają dwa dodatkowe rejestry segmentowe które możemy używać w programach użytkowych. Oprócz tej poprawy ,którą każdy program może wykorzystać, procesory 80386/486 mają również kilka dodatkowych rejestrów z poziomu systemu dla zarządzania pamięcią, uruchomieniowych i testujących procesor .Zobacz

- Rejestry 80386/486

Rodzina 80x86 Intela używa rozbudowanych schematów adresowania pamięci, znanych jako adresowanie segmentowe które dostarcza symulowanych dwuwymiarowych adresów .Pozwala to nam zgrupować logicznie pokrewne bloki danych wewnątrz segmentu. Dokładny format tych segmentów zależy od tego czy CPU działa w trybie rzeczywistym czy chronionym. Większość programów DOSowskich działa w trybie rzeczywistym. Kiedy pracujemy w trybie rzeczywistym, bardzo łatwo jest konwertować logiczny(segmentowy) adres do

liniowego fizycznego adresu. Jednak, w trybie chronionym taka konwersja jest znacznie bardziej utrudniona. Zobacz:

- Segmenty w 80x86

Z powodu sposobu odwzorowania adresu segmentowego do adresu fizycznego w trybie rzeczywistym ,jest całkiem możliwe mieć dwa różne adresy segmentowe które odnoszą się do tej samej komórki pamięci .Jednym rozwiązaniem tego problemu jest użycie adresu znormalizowanego .Jeśli dwa adresy znormalizowane ie mają tego samego wzoru bitów, wskazują różne adresy. Znormalizowane wskaźniki są używane kiedy porównujemy wskaźniki w trybie rzeczywistym .Zobacz:

- Znormalizowane adresy w 80x86

Z wyjątkiem dwóch instrukcji,80x86 nie pracuje właściwie z pełnym 32 bitowym dresem segmentowym. Zamiast tego, używa rejestrów segmentowych do przechowania domyślnej wartości segmentu. Pozwoliło to projektantom z Intela zbudować dużo mniejszy zbiór instrukcji ponieważ adresy są tylko 16 bitowe(tylko część offsetowa) zamiast 32 bitowej długości.80286 i wcześniejsze procesory zawierają cztery rejestry segmentowe: cs,ds.,es iss;80386 i późniejsze procesory dostarczają sześć rejestrów segmentowych: cs,ds.,es,fs,gs i ss .Zobacz:

- Rejestry segmentowe w 80x86

Rodzina 80x86 dostarcza wiele różnych sposobów dostępu do zmiennych, stałych i innych danych. Nazwa dla mechanizmu przez który uzyskujemy dostęp do komórki pamięci to tryb adresowania. Procesory 8088,8086 i 80286 dostarczają dużego zbioru trybów adresowania pamięci. Zobacz:

- Tryby adresowania 80x86
- Tryb adresowania rejestrów 8086
- Tryb adresowania pamięci 8086

Procesor 80386 i późniejsze dostarczają rozszerzony zbiór trybów adresowania rejestrów i pamięci. Zobacz:

- Tryby adresowania rejestrów 80386
- Tryby adresowania pamięci 80386

Większość powszechnych instrukcji 80x86 to instrukcje mov.Ta instrukcja wspiera większość trybów adresowania dostępnych w rodzinie procesorów 80x86.Dlatego też, instrukcja mov jest dobrą instrukcją kiedy studiujemy kodowanie i działanie instrukcji 80x86.Zobacz:

- Instrukcja MOV 80x86

Instrukcja mov przybiera kilka ogólnych form, pozwalających nam przenosić dane między rejestrem a inną lokacją. Możliwe lokacje źródło / przeznaczenie zawiera: (1) inne rejestry,(2) komórki pamięci (używając generalnego trybu adresowania pamięci),(3) stałe (używając trybu adresowania natychmiastowego) i (4) rejestry segmentowe.

Instrukcja mov pozwala przenosić dane między dwoma komórkami (chociaż nie możemy przenosić danych między dwoma komórkami pamięci)

4.12 PYTANIA

1) Chociaż procesory zawsze używają adresowania segmentowego, kodowanie instrukcji dla instrukcji ,takiej jak „mov AX,I" ma tylko 16 bitowy offset kodowany w opcodzie .Wyjaśnij
2) Adresowanie segmentowe jest najlepiej opisane jako schemat dwuwymiarowego adresowania. Wyjaśnij.
3) Skonwertuj następujące adresy logiczne do adresów fizycznych Załóż wszystkie wartości jako heksadecymalne i działanie w trybie rzeczywistym w 80x86:
   a) 1000:1000 b) 1234:5678 c) 0:1000 d) 100:9000 e) FF00:1000
   f) 800:8000 g) 8000:800 h) 234:9843 i) 1111:FFFF j) FFFF:10
4) Doprowadź powyższe adresy do postaci znormalizowanej
5) Wymień wszystkie tryby adresowania pamięci w 80x86
6) Wymień wszystkie tryby adresowania w 80386 (i późniejszych) które nie są dostępne w 8086 (uzyj ogólnej formy jak disp[reg]),nie wyliczaj wszystkich możliwych kombinacji.
7) Oprócz trybu adresowania pamięci jakie inne dwa główne tryby adresowania są w 8086
8) Opisz powszechne użycie dla każdego z następujących trybów adresowania:
   a) rejestrów
   b) Tylko przemieszczenie
   c) Natychmiastowy
   d) Bezpośredni przez rejestr
   e) Indeksowany
   f) Bazowy indeksowany
   g) Bazowy indeksowany z przemieszczeniem
   h) Indeksowany ze skalowaniem

9) Podaj wzorzec bitów dla generowania instrukcji MOV (zobacz „Instrukcja MOV 80x86”),wyjaśnij dlaczego 80x86 nie wspiera operacji przesunięcia z pamięci do pamięci
10) Która z następujących instrukcji MOV nie jest obsługiwana przez opcod ogólnej instrukcję MOV Wyjaśnij.
    a) mov ax,bx b)mov ax,1234 c)mov ax,1 d) mov ax,[bx] e) mov ax, ds. f) mov [bx],2
11) Załóżmy ,że zmienna „I” jest offsetem 20h w segmencie danych .Dostarcz kodowania binarnego dla powyższych instrukcji
12) Co określa ,że pole R/M. specyfikuje rejestr albo pamięć jako operand?
13) Jakie pole w bicie REG-MOD-R/M. określa rozmiar przemieszczenia następujących instrukcji? Jaki rozmiar przemieszczenia wspiera 8086?
14) Dlaczego tryb adresowania „tylko z przemieszczeniem” nie wspiera wielokrotnego przemieszczenia rozmiarów?
15) Dlaczego nie chcielibyśmy zamieniać dwóch instrukcji „mov ax,[bx]” i „mov ax,[ebx}”?
16) Pewne instrukcje 80x86 przybierają kilka postaci .Na przykład, są dwie różne wersje instrukcji MOV, która ładuje rejestr wartością natychmiastową. Wyjaśnij dlaczego projektanci włączyli tą nadmiarowość do zbioru instrukcji?
17) Dlaczego nie jest to prawdziwy tryb adresowania [bp]?
18) Wymień wszystkie ośmiobitowe rejestry 80x86.
19) Wymień wszystkie 16 bitowe rejestry ogólnego przeznaczenia.
20) Wymień wszystkie rejestry segmentowe (te dostępne na wszystkich procesorach)
21) Opisz „specjalne przeznaczenie” każdego z rejestrów ogólnego przeznaczenia.
22) Wymień wszystkie 32 bitowe rejestry ogólnego przeznaczenia 80386/486/586
23) Jakie są związki pomiędzy 8,16 i 32 bitowymi rejestrami ogólnego przeznaczenia w 80386?
24) Jakie wartości pojawiają się w rejestrze flag 8086?Rejestrze flag 80286?
25) Które flagi są kodami stanu?
26) Który rejestr ekstra segmentu pojawia się w 80386 ale nie we wcześniejszych procesorach?