ROZDZIAŁ SZÓSTY:**ZBIÓR INSTRUKCJI PROCESORA 80x86**

Dotychczas ,tylko trochę omówiliśmy dostępne instrukcje w mikroprocesorze 80x86.Ten rozdział naprawi tą sytuację. Zauważmy, że ten rozdział jest głównie odniesieniem .Wyjaśnia co każda instrukcja robi, nie wyjaśnia jak łączą się te instrukcje w programie napisanym w języku asemblera. Reszta tej książki wyjaśnia jak się to robi.

6.0 WSTĘP

Ten rozdział omawia zbiór instrukcji trybu rzeczywistego 80x86.Podobnie jak w innych językach programowania, będzie kilka instrukcji, których będziemy używać cały czas ,kilka będziemy używać okazjonalnie ,a kilku będziemy używać rzadko, jeśli w ogóle. Rozdział ten będzie przedstawiał instrukcje według klas zamiast według znaczenia. Ponieważ początkujący programiści asemblerowi nie muszą uczyć się całego zbioru instrukcji żeby pisać sensowne programy asemblerowe ,nie będziemy się musieli uczyć jak każda instrukcja działa. Następująca lista opisuje instrukcje omawiane w tym rozdziale. Symbol „•" oznacza ważne instrukcje w każdej grupie. Jeśli nauczymy się tylko tych instrukcji, będziemy mogli napisać każdy program asemblerowy jaki zechcemy .Jest wiele dodatkowych instrukcji, zwłaszcza w procesorze 80386 i późniejszych .Te dodatkowe instrukcje czynią programowanie w asemblerze łatwiejszym, ale nie musimy ich znać aby zacząć pisać programy.

Instrukcje 80x86 mogą być (z grubsza) podzielone na osiem różnych klas:

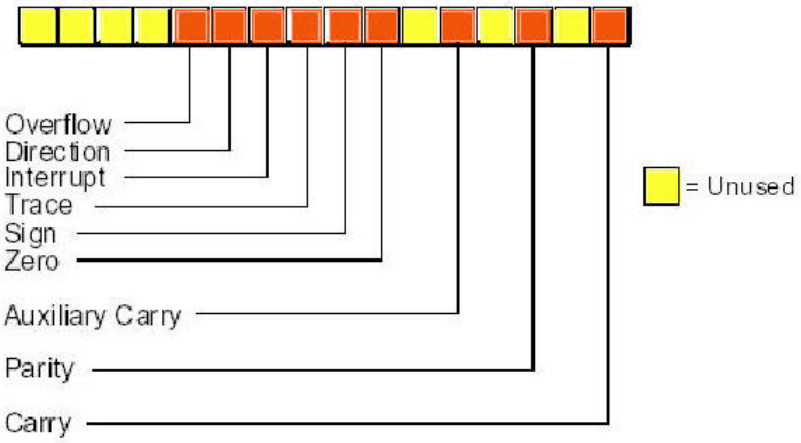
1) Instrukcje przenoszenia danych
   - mov, lea, les, push, pop ,pushf, popf
2) Konwersja
   - cbw, cwd, xlat
3) Instrukcje arytmetyczne
   - Add ,inc ,sub ,dec ,cmp ,neg ,mul, imul, div ,idiv
4) Instrukcje logiczne ,przesunięcia, obrotu i bitowe
   - and, or, xor, not, shl, shr ,rcl ,rcr
5) Instrukcje I/O
   - in, out
6) Instrukcje łańcuchowe
   - movs, stos ,lods
7) Instrukcje sterowania strumieniem danych w programie
   - jmp, call, ret, skoki warunkowe
8) Instrukcje różne
   - clc, stc, cmc

Następna sekcja omówi wszystkie te instrukcje w tych grupach i jak one działają.

Raz jeden jeszcze w tym tekście nie rekomenduje się używania rozszerzonego zbioru instrukcji 80386.Program,który używa takich instrukcji może nie pracować właściwie na procesorze 80286 lub wcześniejszych. Używanie tych dodatkowych instrukcji może ograniczyć liczbę maszyn na których nasz kod będzie działał. Jednak, procesor 80386 zostanie w tym tekście opisany. Możemy bezpiecznie założyć, że większość systemów będzie zawierać procesor 80386sx lub późniejsze. Ten tekst często używa zbioru instrukcji 80386 w różnych przykładowych programach. Zapamiętajmy, jest to zrobione tylko dla wygody .Nie ma

programu który pojawia się w tym tekście, który nie mógłby być przekodowany przy użyciu tylko instrukcji asemblerowych CPU 8088.

Słowo porady, szczególnie dla tych, którzy uczą się tylko powyższych instrukcji: jeśli czytasz o zbiorze instrukcji 80x86,odkryjesz,że pojedyncze instrukcje 80x86 nie są bardzo złożone i mają prostą semantykę. Jednak jeśli nadejdzie



Rysunek 6.1 :Rejestr flag 80x86

koniec tego rozdziału, możesz odkryć ,że nie masz pojęcia jak ułożyć te proste instrukcje razem w formie złożonego programu. Nie bój nic, jest to powszechny problem. Późniejsze rozdziały omówią jak sformować złożony program z tych prostych instrukcji.

Jedna szybka uwaga: ten rozdział wyliczył dużo instrukcji jakie „dostępne są na procesorze 80286 i późniejszych procesorach". Faktycznie wiele z tych instrukcji było również dostępnych w mikroprocesorze 80186.Ponieważ tylko kilka systemów PC stosuje mikroprocesor 80186,ten tekst ignoruje ten CPU

6.1 REJESTRY STANU PROCESORA (FLAGI)

Rejestr flag utrzymuje bieżący tryb działania CPU i informuje o jego stanie. Rysunek 6.1 pokazuje układ rejestru flag.

Znaczniki przeniesienia, parzystości, zera, znaku i przepełnienia są specjalne ponieważ możemy testować ich stan (zero lub jeden) setcc i instrukcjami skoków warunkowych (zobacz „Zbiór instrukcji warunkowych" i „Instrukcje skoków warunkowych") 80x86 używa tych bitów ,kodów błędu, do podjęcia decyzji podczas wykonywania programu.

Różne arytmetyczne ,logiczne i inne różne instrukcje wpływają na znacznik przepełnienia (overflow).Po operacjach arytmetycznych, flaga ta zawiera jeden jeśli wynik nie mieści się w operandzie przeznaczenia ze znakiem. Na przykład, jeśli próbujemy dodawać 16 bitową liczbę ze znakiem 7FFFh i 0001h,wynik jest zbyt duży więc CPU ustawia flagę przepełnienia. Jeśli wynik operacji arytmetycznej nie tworzy przepełnienia, wtedy CPU czyści tą flagę.

Ponieważ generalnie operacje logiczne stosują wartości bez znaku, instrukcje logiczne 80x86 po prostu czyszczą flagę przepełnienia. Inne instrukcje 80x86 opuszczają flagę przepełnienia zawierającą dowolna wartość.

Instrukcje łańcuchowe 80x86 używają znacznika kierunku (direction flag).Kiedy flaga kierunku jest wyczyszczona, 80x86 przetwarza elementy łańcucha od adresu niższego do wyższego; kiedy ustawiona CPU przetwarza łańcuch w kierunku przeciwnym. Zobacz "Instrukcje Łańcuchowe: po dodatkowe szczegóły.

Znacznik zezwolenia na przerwanie (interrupt enable/disable flag) steruje zdolnością 80x86 do odpowiedzi na zewnętrzne zdarzenie znane jako prośba o przerwanie. Niektóre programy zawierają pewną sekwencję instrukcji, których nie wolno przerwać CPU. Flaga zezwolenia na przerwanie włącza lub wyłącza przerwania gwarantując, że CPU nie przerwie tej krytycznej sekcji kodu.

Znacznik stanu śledzenie (trace flag) włącza lub wyłącza tryb śledzenia. Debuggery (takie jak CodeView) używają tego bitu do włączania lub wyłączania operacji pojedynczego kroku śledzenia. Kiedy jest ustawiony, CPU przerywa każdą instrukcję i przekazuje sterowanie do debuggera pozwalając mu na wykonywanie pojedynczego kroku przez aplikację. Jeśli ten bit jest wyczyszczony ,wtedy 80x86 wykonuje instrukcje bez przerwania. CPU 80x86 nie dostarcza żadnej instrukcji ,która bezpośrednio manipuluje tą flagą .Aby ustawić lub wyczyścić tą flagę musimy:

- Odłożyć flagi na stos 80x86
- Ściągnąć wartość innego rejestru
- Ustawić wartość flagi stanu śledzenia

- Odłożyć wynik na stos a potem
- Ściągnąć flagi ze stosu

Jeśli wynik jakiegoś obliczenia jest negatywany,80x86 ustawia znacznik znaku. Możemy przetestować tę flagę po operacji arytmetycznej dla sprawdzenia ujemnego wyniku. Pamiętamy ,wartość jest ujemna jeśli jej najbardziej znaczący bit wynosi jeden. Dlatego też operacje na wartościach bez znakowych ustawią flagę znaku jeśli rezultat ma jedynkę na najbardziej znaczącej pozycji.

Różne instrukcje ustawiają znacznik zera kiedy generują jako wynik zero. Często będziemy używać tej flagi aby zobaczyć, czy dwie wartości są równe (np. po odjęciu dwóch liczb, są one równe jeśli wynik wynosi zero).Flaga ta jest również użyteczna po różnych operacjach logicznych aby zobaczyć czy wyspecyfikowany bit w rejestrze lub pamięci zawiera zero czy jeden.

Znacznik przeniesienia połówkowego wspiera operacje specjalnego systemu dziesiętnego kodowanego dwójkowo (BCD).Ponieważ większość programów nie stosuje liczb BCD, rzadko będziemy używać tej flagi a i nawet wtedy nie uzyskamy do niej bezpośrednio dostępu.. CPU 80x86 nie dostarczają żadnej instrukcji, która pozwalałaby nam bezpośrednio testować, ustawiać i czyścić tą flagę. Tylko instrukcje add, adc, sub, sbb, mul, imul, div, idiv i BCD manipulują tą flagą.

Znacznik parzystości jest ustawiany według parzystości najmniej znaczących bitów każdej operacji na danych. Jeśli operacja tworzy parzystą liczbę bitów, CPU ustawia tą flagę. Zeruje tą flagę jeśli operacja przynosi nieparzystą liczbę bitów. Flaga ta jest użyteczna w pewnych programach komunikacyjnych, jednak Intel wprowadził go głównie dla kompatybilności ze starszymi mikroprocesorami 8080.

Znacznik przeniesienia ma kilka zadań. Po pierwsze oznacza przepełnienie bez znakowe (podobnie jak znacznik przepełnienia wykrywa przepełnienie znakowe).Możemy go również użyć podczas operacji arytmetycznych i logicznych o dużej dokładności. Pewne bity testowe, ustawiania, zerowania i inwersji w 80386 bezpośrednio wpływają na tą flagę. W końcu, ponieważ możemy łatwo zerować, ustawiać ,odwracać i testować ją, jest użyteczna dla różnych operacji boolowskich. Flaga przeniesienia ma wiele zadań i znajomość kiedy ją użyć i w jakim celu może wprowadzić w zakłopotanie początkującego programistę asemblerowego. Na szczęście, dla większości danych instrukcji, flaga przeniesienia jest zerowana.

Używanie tych znaczników stanie się łatwe i oczywiste w przyszłych sekcjach i rozdziałach. Ta sekcja jest głównie formalnym wprowadzeniem do pojedynczych flag w rejestrze zamiast próba dokładnego wyjaśnienia funkcjonowania każdej z flag.

6.2 KODOWANIE INSTRUKCJI

80x86 używa kodowania binarnego dla każdej operacji maszynowej. Podczas gdy jest ważne mieć ogólne pojęcie jak 80x86 koduje instrukcje, nie jest tak ważne wprowadzanie do pamięci kodowania dla wszystkich instrukcji w zbiorze instrukcji. Gdybyśmy pisali asembler lub disasembler (debugger) musielibyśmy koniecznie znać dokładnie kodowanie .Jednak dla ogólnych programów asemblerowych nie musimy znać dokładnego kodowania.

Ponieważ stajemy się bardziej doświadczeni w asemblerze prawdopodobnie chcielibyśmy studiować kodowania zbioru instrukcji 80x86.Oczywiście powinniśmy być zaznajomieni z takimi terminami jak opcod ,bajt mod-reg-r/m., wartość przemieszczenia ,i innymi. Chociaż nie musimy zapamiętywać parametrów dla każdej instrukcji, jest zawsze dobrym pomysłem znać długość i czas cyklu dla instrukcji używanych regularnie ponieważ pomoże to nam pisać lepsze programy. Rozdział Trzeci i Rozdział Czwarty dostarczyły szczegółów odnośnie kodowania instrukcji dla różnych instrukcji (80x86 i x86);takie omówienie było ważne ponieważ musimy zrozumieć jak CPU koduje i wykonuje instrukcje. ten rozdział nie zajmuje się takimi szczegółami Ten rozdział przedstawia wysokopoziomowy obraz każdej instrukcji i zakłada, że nie interesuje nas jak maszyna traktuje bity w pamięci. Dla tych kilku razy, kiedy będziemy musieli poznać kodowanie binarne dla poszczególnych instrukcji, kompletny listing kodowania instrukcji zawiera Appendix D.

6.3 INSTRUKCJE PRZENOSZENIA DANYCH

Instrukcje przenoszenia danych kopiują wartości z jednej lokacji do innej. Te instrukcje to mov,xchg,lds,lea,les,lfs,lgs,lss,push,pusha,pushad,pushf,pushfd,pop,popa,popad,popf,popfd,lahf i sahf

6.3.1 INSTRUKCJA MOV

Instrukcja mov przybiera kilka różnych form:

```
mov     reg, reg
mov     mem ,reg
mov     reg, mem
mov     mem, dana bezpośrednia
mov     reg, dana bezpośrednia
mov     ax/al., mem
mov     mem, ax/al.
```

```
mov     segreg, mem16
mov     segreg, reg16
mov     mem16, segreg
mov     reg16 ,segreg
```

Ostatni rozdział omawiał instrukcję mov szczegółowo, więc tylko trochę komentarzy godnych uwagi .Po pierwsze, są to warianty instrukcji mov które są szybsze i krótsze niż inne instrukcje mov wykonujące tą samą pracę. Na przykład, obie instrukcje mov ax, mem i mov reg, mem mogą załadować rejestr ax z komórki pamięci. We wszystkich procesorach, pierwsza wersja jest krótsza .We wcześniejszych członkach rodziny 80x86 jest również szybsza.

Są dwa bardzo ważne szczegóły dotyczące instrukcji mov. Po pierwsze nie ma operacji przeniesienia z pamięci do pamięci. tryb adresowania bajt mod-reg-r/m. (zobacz Rozdział Czwarty) pozwala na to dwóm operandom rejestrowym lub jednemu operandowi rejestrowemu i jednemu operandowi pamięci.. nie ma formy instrukcji mov, która pozwala nam kodować dwa adresy pamięci w tej samej instrukcji. Po drugie, nie możemy bezpośrednio przenieść danych do rejestru segmentowego. Instrukcje które przenoszą dane do lub z rejestru segmentowego mają bajty mod-reg-r/m. ;nie ma formy która przenosi bezpośrednie wartości do rejestru segmentowego .Dwa powszechne błędy robione przez początkujących programistów to próby przenoszenia z pamięci do pamięci i próby ładowania stałej do rejestru segmentowego.

Operandami instrukcji mov mogą być bajty, słowa lub podwójne słowa. Oba operandy muszą być tego samego rozmiaru lub MASM wygeneruje błąd podczas asemblowania naszego programu. To ma zastosowanie do operandu pamięci i rejestru. Jeśli deklarujemy zmienną B, używając byte i próbujemy załadować tą zmienną do rejestru ax ,MASM zgłosi konflikt typów.

CPU rozszerza dane bezpośrednie do rozmiaru operandu przeznaczenia (chyba ,że jest zbyt duży do umieszczenia w operandzie przeznaczenia, wtedy mamy błąd).Zauważmy, że możemy przenieść wartości bezpośrednie do komórki pamięci. Ta sama zasada dotyczy stosowanego rozmiaru. Jednak MASM nie może ustalić rozmiaru pewnych operandów pamięci. Na przykład, czy instrukcja mov [bx],0 przechowuje wartość ośmio- szesnasto- czy trzydziesto dwu bitową? MASM nie może powiedzieć wiec zgłasza błąd. Ten problem nie istnieje, jeśli przenosimy dane do zmiennych, zadeklarowanych w naszym programie. Na przykład, jeśli zadeklarujemy B jako zmienną bajtową, MASM wie, że ma przechować osiem bitów zero w B dla instrukcji mov B,0.Tylko te operandy pamięci wymagające wskaźników bez zmiennych operandów cierpią na ten problem. Rozwiązaniem jest wyraźne powiedzenie MASMowi czy operand jest bajtem, słowem czy podwójnym słowem, Możemy tego dokonać w następujących formach instrukcji:

```
mov     byte ptr [bx],0
mov     word ptr [bx],0
mov     dword ptr [bx],0    (3)
```

(3) dostępne tylko na procesorach 80386 i późniejszych

Po więcej szczegółów na temat operatora type ptr zajrzyj do Rozdziału Ósmego.

Przeniesienia do i z rejestru segmentowego są zawsze 16 bitowe; operand mod-reg-r/m. musi być 16 bitowy albo MASM wygeneruje błąd. Ponieważ nie możemy załadować stałej bezpośrednio do rejestru segmentowego ,popularnym rozwiązaniem jest ładowanie stałej do rejestru ogólnego przeznaczenia a potem skopiowanie go do rejestru segmentowego. Na przykład, następujące dwie instrukcje ładują rejestr es wartością 40h:

```
mov     ax,40h
mov     es, ax
```

Zauważmy ,że prawie każdy rejestr ogólnego przeznaczenia jest wystarczający. Tutaj, ax został wybrany przypadkowo.

Instrukcja mov nie wymaga żadnych flag. W szczególności,80x86 zachowuje wartości flag po wykonaniu instrukcji mov.

6.3.2 INSTRUKCJA XCHG

Instrukcja xchg (exchange) zamienia miejscami dwie wartości. Ogólna jej forma to

```
xchg        operand1, operand2
```

Są cztery specyficzne formy tej instrukcji w 80x86:

```
xchg        reg, mem
xchg        reg, reg
xchg        ax, reg16
xchg        eax, reg32      (3)
```

(3) dostępne tylko na procesorze 80386 i późniejszych

Pierwsze dwie formy ogólne wymagają dwóch lub więcej bajtów na opcody i bajty mod-reg-r/m.(przemieszczenie, jeśli jest konieczne ,wymaga dodatkowego bajtu).Trzecia i czwarta forma są specjalnymi

formami drugiej, która zamienia dane w rejestrze (e)ax z innym 16- lub 32 bitowym rejestrem. Forma 16 bitowa używa pojedynczego bajtu opcodu, który jest krótszy niż pozostałe dwie formy, które używają jednego bajtu opcodu i bajtu mod-reg-r/m.

Powinniśmy zanotować wzorzec konstruowania: rodzina 80x86 często dostarcza krótszych i szybszych wersji instrukcji ,które używają rejestry ax. Dlatego też, powinniśmy próbować tak układać swoje obliczenia aby używać rejestru (e)ax tak często jak to możliwe. Instrukcja xchg jest doskonałym przykładem, forma, która zamienia 16 bitowe rejestry jest długa na jeden bajt.

Zauważmy, że porządek operandów xchg nie ma znaczenia. To znaczy możemy wprowadzić xchg mem, reg i otrzymać ten sam wynik jak przy xchg reg, mem. Większość nowoczesnych asemblerów automatycznie emituje opcod dla krótszej instrukcji xchg ax, reg jeśli wyszczególnimy xchg reg, ax.

Oba operandy muszą być tego samego rozmiaru .Na procesorach przed 80386, operandy mogą być ośmio- lub szesnasto bitowe. Na procesorach 80386 i późniejszych operandy mogą być długie na 32 bity.

Instrukcja xchg nie modyfikuje żadnych flag.

6.3.3 INSTRUKCJE LDS,LES,LFS,LGS I LSS

Instrukcje lds ,les, lfs, lgs i lss pozwalają nam załadować 16 bitowy rejestr ogólnego przeznaczenia i rejestr segmentowy pojedyncza instrukcją W 80296 i wcześniejszych, instrukcje lds i les są instrukcjami, które bezpośrednio przetwarzają wartości większe iż 32 bity. Ogólna forma:

LxS prez. , źródło

Te instrukcje przyjmują takie formy:

lds	reg_{16}, mem_{32}	
les	reg_{16}, mem_{32}	
lfs	reg_{16}, mem_{32}	(3)
lgs	reg_{16}, mem_{32}	(3)
lss	reg_{16}, mem_{32}	(3)

(3) dostępne na procesorach 80386 lub późniejszych

Reg_{16} jest rejestrem ogólnego przeznaczenia a mem_{32} jest komórką pamięci podwójnego słowa (zadeklarowaną instrukcją dword).

Te instrukcje ładują 32 bitowe spod adresu wyspecyfikowanego przez mem_{32} do reg_{16} i rejestrów ds., es, fs, gs lub ss. ładują one rejestr ogólnego przeznaczenia najmniej znaczącym słowem operandu pamięci a rejestr segmentowy słowem najbardziej znaczącym. Następujący algorytm opisuje dokładnie tą operację:

```
lds reg16, mem32;
                reg16 := [mem32]
                ds := [mem32 + 2]
les reg16, mem32;
                reg16 := [mem32]
                es := [mem32 + 2]
lfs reg16, mem32;
                reg16 := [mem32]
                fs := [mem32 + 2]
lgs reg16, mem32;
                reg16 := [mem32]
                gs := [mem32 + 2]
lss reg16, mem32;
                reg16 := [mem32]
                ss := [mem32 + 2]
```

Ponieważ instrukcje LxS ładują rejestry segmentowe 80x86,nie wolno nam używać tych instrukcji dla przypadkowych zadań. Użycie ich do ustawienia (dalekich) wskaźników do pewnych obiektów danych jest omówione w Rozdziale Czwartym. Inne ich zastosowanie może spowodować problemy z naszym kodem jeśli spróbujemy przenieść go na Windows,OS/2 lub UNIX.

Zapamiętajmy, że te instrukcje ładują cztery bajty spod danej komórki pamięci do pary rejestrów; one nie ładują adresów zmiennych do pary rejestrów (tj. ta instrukcja nie ma trybu bezpośredniego).Jeśli chcemy się nauczyć jak ładować adresy zmiennych do pary rejestrów ,zobaczymy to w Rozdziale Ósmym.

Instrukcje LxS nie wpływają na żaden bit flagi 80x86.

6.3.4 INSTRUKCJA LEA

Instrukcja LEA (Załaduj adres efektywny) jest inną instrukcją używaną do przygotowania wartości wskaźnika. Instrukcja lea przybiera formę:

```
lea     prez. , źródło
```

Formy w 80x86 to:

```
lea     reg16, mem
lea     reg32, mem      (3)
```

(3) dostępne tylko na procesorach 80386 lub późniejszych

Ładuje ona określony 16 lub 32 bitowy rejestr ogólnego przeznaczenia adresem efektywnym określonej komórki pamięci. Adres efektywny jest końcowym adresem pamięci uzyskanym w wyniku końcowego obliczania trybu adresowania .na przykład, lea ax, ds:[1234h] ładuje rejestr ax adresem komórki pamięci 1234h;tu jest ładowna do rejestru ax wartość 1234h Jeśli pomyślimy o tym przez chwilę, to nie jest bardzo ekscytująca operacją końcu instrukcja mov ax, dana_ bezpośrednia może to zrobić. Więc dlaczego zawracamy sobie głowę instrukcją lea? Cóż, jest wiele innych form operandów pamięci poza operandem „tylko przemieszczenie". Rozważmy następujące instrukcje lea:

```
lea     ax, [bx]
lea     bx, 3[bx]
lea     ax, 3[bx]
lea     bx, 4[bp+si]
lea     ax, -123[di]
```

Instrukcja lea ax, [bx] kopiuje adres wyrażenia [bx] do rejestru ax. Ponieważ adres efektywny jest wartością w rejestrze bx, ta instrukcja kopiuje wartość bx do rejestru ax .Znów, ta instrukcja nie jest bardzo interesująca ponieważ mov może robić to samo ,nawet szybciej.

Instrukcja lea bx,3[bx] kopiuje adres efektywny z 3[bx] do rejestru bx. Ponieważ adres efektywny jest równy bieżącej wartości z bx plus trzy, ta instrukcja lea skutecznie doda trzy do rejestru bx Jest instrukcja add, która pozwala nam dodać trzy do rejestru bx, więc znowu, instrukcja lea jest zbyteczna do tego celu.

Trzecia instrukcja lea pokazuje gdzie lea rzeczywiście zaczyna błyszczeć. Lea ax, 3[bx] kopiuje adres z komórki pamięci 3[bx] do rejestru ax; tj. dodaje trzy do wartości w rejestrze bx i przenosi tą sumę do ax. Jest to doskonały przykład jak można użyć instrukcji lea do operacji mov i w dodatku w pojedynczej instrukcji.

Dwie końcowe instrukcje, lea bx,4[bp+si} i lea ax, -123[di] dostarczają dodatkowych przykładów instrukcji lea, które są bardziej wydajne niż ich odpowiedniki mov /add.

W 80386 i późniejszych procesorach, może używać trybu adresowania indeksowego ze skalowaniem do mnożenia przez dwa, cztery lub osiem jak również dodać rejestry i przemieszczenie razem Intel zdecydowanie sugeruje używanie instrukcji lea ponieważ jest ona dużo szybsza niż sekwencja instrukcji obliczająca ten sam rezultat.

(Rzeczywistym) celem lea jest ładowanie rejestru adresem pamięci. Na przykład, lea bx, 128[bp+di] ustawia bx adresem bajtu odnoszącym się do bajtu 128[bp+di].Okazuje się, że instrukcja w formie mov al.,[bx] wykonuje się szybciej niż instrukcja mov al.,128[bp+di].Jeśli ta instrukcja wykona się kilka razy, będzie wydajniej załadować adres efektywny ze 128[bp+di] do rejestru bx i użycie trybu adresowania [bx].Jest to powszechna optymalizacja w wysoko wydajnych programach.

Instrukcja lea nie wpływa na żaden bit flag 80x86.

6.3.5 INSTRUKCJE PUSH I POP

Instrukcje 80x86 PUSH i POP manipulują danymi na stosie sprzętowym 80x86.Jest 19 wariantów instrukcji push i pop, oto i one

```
push    reg16
pop     reg16
push    reg32                   (3)
pop     reg32                   (3)
push    segreg
pop     segreg                  (except CS)
push    memory
pop     memory
push    immediate_data          (2)
pusha                           (2)
popa                            (2)
pushad                          (3)
popad                           (3)
pushf
popf
pushfd                          (3)
popfd                           (3)
enter   imm, imm                (2)
leave                           (2)
```

(2) dostępne tylko na procesorach 80286 i późniejszych
(3) dostępne na procesorach 80386 i późniejszych

Pierwsze dwie instrukcje kładą i zdejmują 16 bitowy rejestr ogólnego przeznaczenia. Jest to małych rozmiarów (jedno bajtowa) wersja stworzona specjalne dla rejestrów. Zauważmy, że jest druga forma która dostarcza bajtu mod-reg-r/m., która może kłaść rejestry również ;większość asemblerów używa tylko tamtej formy dla położenia wartości komórki pamięci.

Druga para instrukcji kładzie i zdejmuje 32 bitowe rejestry ogólnego przeznaczenia 80386.Jest to rzeczywiście nic więcej niż położenie rejestru instrukcją opisaną w poprzednim paragrafie którego przyrostek ma rozmiar bajta.

Trzecia para instrukcji push/pop pozwala nam położyć i zdjąć rejestry segmentowe 80x86.Zauwazmy,że instrukcje takie jak push fs i gs są dłuższe niż te które kładą cs, ds., es i ss. Zobacz Appendix D po więcej szczegółów ,Możemy także położyć rejestr cs (zdjęcie ze stosu rejestru cs co stwarzałoby pewne interesujące problemy kontroli przepływu programu).

Czwarta para instrukcji push/pop pozwala nam położyć lub zdjąć zawartość komórki pamięci. W 80286 i wcześniejszych ,musi to być wartość 16 bitowa. Dla operacji pamięci bez wyraźnego typu (np. [bx]) musimy użyć albo mnemonika pushw albo jawnego stanu rozmiaru używanej instrukcji jak push word ptr [bx].W 80386 i późniejszych możemy położyć i zdjąć wartości 16 i 32 bitowe. Możemy użyć operandu pamięci dword, mnemonika pushd lub operator dword ptr do wymuszenia operacji 32 bitowej.

Przykłady:

```
push    DblWordVar
push    dword ptr [bx]
push    dword
```

Instrukcje pusha i popa (dostępne na 80286 i późniejszych) kładą i zdejmują wszystkie 16 bitowe rejestry ogólnego przeznaczenia. Pusha kładzie rejestry w następującym porządku: ax, cx, dx, bx, sp ,bp ,si a potem di. Popa zdejmuje te rejestry w porządku odwrotnym. Pushad i Popad (dostępne na 80386 i późniejszych) robią to samo na zbiorze 32 bitowych rejestrów 80386.Zauważmy,że te „kładące wszystko" i zdejmujące wszystko" instrukcje nie kładą ani nie zdejmują znaczników ani rejestrów segmentowych.

Instrukcje pushf i popf pozwalają nam kłaść / zdejmować rejestr stanu procesora (flagi).Zauważmy, że te dwie instrukcje dostarczające mechanizm do modyfikacji znacznika śledzenia stanu. Zobacz opis działania wcześniej w tym rozdziale. Oczywiście możemy ustawić lub wyczyścić tym sposobem również inne flagi. Jednakże, większość innych flag chcielibyśmy modyfikować (kody błędów) dostarczając określonych instrukcji lub innych prostych sekwencji do tego celu.

Enter i leave kładą / zdejmują rejestr bp i alokują pamięć dla lokalnych zmiennych na stosie .Dowiemy się więcej o tych instrukcjach w późniejszym rozdziale. Ten rozdział ich nie rozpatruje ponieważ nie są one szczególnie użyteczne w oderwaniu od wejścia i wyjścia z procedury.

„Więc co robią te instrukcje?" spytamy prawdopodobnie. Instrukcje push przenoszą dane na stos 80x86 a instrukcje pop przenoszą dane ze stosu do pamięci lub rejestru. Poniżej znajduje się opisany algorytm każdej instrukcji:

```
instrukcja push (16 bitowa):
        SP:=Sp-2
        [SS:SP]:= operand 16 bitowy (przechowuje wynik w lokacji SS:SP)
instrukcja pop (16 bitowa):
        operand 16 bitowy:=[SS:SP]
        SP:=SP+2
instrukcja push (32 bitowa):
        SP"=SP-4
        [SS:SP]:= 32 bitowy operand
instrukcja pop (32 bitowa):
        32 bitowy operand:=[SS:SP]
        SP:=SP+4
```

Możemy potraktować instrukcje pusha/pushad i popa/popad jako równoważniki odpowiadające sekwencji operacji 16 lub 32 bitowych push i pop (np. push ax, push cx, push dx, push bx itd.).

Odnotujmy trzy rzeczy o stosie sprzętowym 80x86.Po pierwsze, jest zawsze w segmencie stosu (gdziekolwiek wskazuje ss).Po drugie, stos maleje w pamięci. To znaczy, jeśli kładziemy wartość na stos CPU przechowuje je w następujących po sobie coraz niższych komórkach pamięci. W końcu, stos sprzętowy 80x86 wskaźnik (ss:sp) zawsze zawiera adres wartości ze szczytu stosu (ostatnia wartość położona na stos).

Możemy użyć stosu sprzętowego 80x86 dla czasowego przechowywania rejestrów i zmiennych, parametrów dla procedur, alokowania pamięci dla lokalnych zmiennych i innych rzeczy .Instrukcje push i pop są niezmiernie cenne dla manipulowania tymi pozycjami na stosie. Dostaniemy szansę zobaczenia jak użyjemy ich później w tym tekście.

Większość instrukcji push i po nie wpływa na żadną flagę stanu rejestru w procesorze 80x86.Instrukcje popf' popfd przez swoją naturę mogą modyfikować wszystkie bity flag rejestru stanu (rejestr flag) procesora 80x86.Pushf i Pushfd odkładają flagi na stos, ale nie zmieniają podczas tego żadnego znacznika.

Wszystkie operacje odkładania i zdejmowania są 16 lub 32 bitowe. Nie ma (łatwego) sposobu włożenia pojedynczej ośmiobitowej wartości na stos. Aby włożyć ośmiobitową wartość będziemy musieli załadować ją do bardziej znaczącego bajtu 16 bitowego rejestru, odłożyć rejestr a potem dodać jeden do wskaźnika stosu. Na wszystkich procesorach z wyjątkiem 8088 spowolniło by to przyszły dostęp do stosu ponieważ sp zawiera teraz nieparzysty adres ,który źle rozmieszczałby dalsze zdjęcia / położenia na stos .Dlatego też, większość programów odkłada lub zdejmuje 16 bitów, nawet jeśli mamy do czynienia z ośmioma bitami.

Chociaż jest stosunkowo bezpiecznie odłożyć osiem bitów zmiennej pamięci, bądźmy ostrożni kiedy zdejmujemy ze stosu do ośmio bitowej komórki pamięci. Odłożenie zmiennej ośmio bitowej push word ptr ByteVar odkłada na stos dwa bajty, bajt zmiennej ByteVar i bajt bezpośredni następujący po niej. Nasz kod może po prostu zignorować ten dodatkowy bajt tej instrukcji odkładany na stos. Zdejmowanie ze stosu takich wartości nie jest całkiem proste. Generalnie ,nie sprawi to trudności jeśli odłożymy te dwa bajty .Jednak, może być nieszczęście jeśli zdejmiemy wartość i zgubimy gdzieś następny bajt w pamięci. Są tylko dwa rozwiązania tego problemu. Po pierwsze, możemy zdjąć wartość 16 bitową do rejestru takiego jak ax a potem przenieść najmniej znaczący bajt tego rejestru do zmiennej bajtowej. Drugim rozwiązaniem jest zarezerwowanie dodatkowego bajtu uzupełniającego po zmiennej bajtowej do przechowania całego słowa które chcemy odłożyć.

6.3.6 INSTRUKCJE LAHF I SAHF

instrukcje LAHF (ładuj ah z flag) i SAHF (przechowaj ah we flagach) są instrukcjami archaicznymi zachowanymi w zbiorze instrukcji 80x86 aby utrzymać zgodność ze starszymi Intelowskimi procesorami typu 8080.jako takie instrukcje te mają bardzo małe zastosowanie w nowoczesnych programach na 80x86.Instrukcja lahf nie wpływa na żaden bit flag. Instrukcja sahf, ze względu na swoją naturę modyfikuje bity S,Z,A,P i C w rejestrze stanu procesora. Te instrukcje nie wymagają żadnego operandu a używamy ich w następujący sposób:

```
sahf
lahf
```

Sahf wpływa tylko na osiem najmniej znaczących bitów rejestru flag. Podobnie lahf, ładuje tylko osiem najmniej znaczących bitów rejestru flag do rejestru AH. Instrukcje te nie wpływają na flagi przepełnienia, kierunku, zabronienia przerwania lub stanu śledzenia. Fakt, że te instrukcje nie wpływają na flagę przepełnienia jest ważnym ograniczeniem.

Sahf ma jedno ważne zastosowanie. Kiedy używamy procesorów zmienno przecinkowych (8087,80287,80387,80487,Pentium itp.) możemy użyć instrukcji sahf do kopiowania zmiennoprzecinkowych flag rejestru stanu do rejestru flag 80x86.Zobaczymy to zastosowanie w rozdziale o arytmetyce zmienno przecinkowej (zobacz :"Arytmetyka Zmiennoprzecinkowa").

6.4 KONWERSJA

Zbiór instrukcji 80x86 dostarcza kilka instrukcji konwersji .Zaliczają się do nich movzx ,movsx, cbw, cwd, cwde, cdq, bswap i xlat. Większość z tych instrukcji stosuje rozszerzenie całkowite (ze znakiem) o zerowe, ostatnie dwie instrukcje konwertują pomiędzy formatami pamięci tłumaczeniem wartości przez tablicę połączeń .Instrukcje te przybierają formę:

```
movzx       przez, źródło    ;przeznaczenie musi być dwa razy większe niż źródło
movsx       przez, źródło    ;przeznaczenie musi być dwa razy większe niż źródło
cbw
cwd
cwde
cdq
bswap       reg 32
xlat
```

6.4.1 INSTRUKCJE MOVZX,MOVSX,CBW,CWD,CWDE I CDQ

Te instrukcje rozszerzają wartości poprzez powielanie zer lub znaków .Instrukcje CBW i CWD są dostępne na procesorach 80x86.Instrukcje movzx, movsx, cwde i cdq są dostępne na procesorach 80386 i późniejszych.

Instrukcja cbw (konwertuj bajt na słowo) powiela znak ośmiobitowej wartości w al. i umieszcza ją w ax. To znaczy kopiuje siódmy bit AL. do bitów 8-15 AX. Ta instrukcja jest ważna zwłaszcza przed wykonaniem ośmiobitowego dzielenia. Ta instrukcja nie wymaga operandów a używamy ją jak następuje:

```
cbw
```

Instrukcja cwd (konwertuj słowo na podwójne słowo) powiela znak 16 bitowej wartości w ax do 32 bitów i umieszcza wynik w dx:ax. kopiuje bit 15 AX do wszystkich bitów dx. jest dostępna na wszystkich procesorach 80x86,co wyjaśnia dlaczego nie powiela znaku wartości do eax. podobnie jak instrukcja cbw, instrukcja ta jest bardzo ważna przy operacjach dzielenia. Nie wymaga operandu a używamy ją :

```
cwd
```

Instrukcja cwde powiela znak 16 bitowej wartości w ax do 32 bitów i umieszcza wynik w eax poprzez kopiowanie bitu 15 do wszystkich bitów 16..31 eax. Instrukcja ta jest dostępna tylko na procesorach 80386 lub późniejszych. Podobnie jak cbw i cwd nie wymaga operandu a używamy ja jak następuje

```
Cwde
```

Instrukcja cdq powiela znak 32 bitowej wartości w eax do 64 bitów i umieszcza wynik w edz:eax przez kopiowanie bitu 31 eax do wszystkich bitów 0..31 edx .Instrukcja ta jest dostępna na procesorach 80386 lub późniejszych. Normalnie będziemy używać tej instrukcji przed operacją długich liczb

Podobnie jak cbw, cwd i cwde nie ma operandu a stosujemy ją tak:

```
Cdq
```

Jeśli chcemy powielić znak wartości ośmiobitowej do 32 lub 64 bitów używając tych instrukcji, możemy zastosować następującą sekwencję:

```
;powielenie znaku al. do dx:ax
        cbw
        cwd
;powielenie znaku al. do eax
        cbw
        cwde
;powielenie znaku al. do edx:eax
        cbw
        cwde
        cdq
```

Możemy również użyć movsx dla powielania znaku z ośmiu do szesnastu lub trzydziestu dwóch bitów.

Instrukcja movsx jest uogólnioną formą instrukcji cbw, cwd i cwde. Powiela znak wartości ośmio bitowej do szesnastu lub trzydziestu dwóch bitów, lub powiela znak wartości szesnasto bitowej do 32 bitów. Ta instrukcja używa bajtu mod-reg-r/m. do wyspecyfikowani dwóch operandów .dostępne formy dla tej instrukcji to

```
movsx   reg16,mem8
movsx   reg16,reg8
movsx   reg32,mem8
movsx   reg32,reg8
movsx   reg32,mem16
movsx   reg32,reg16
```

Zauważmy, że wszystko co możemy zrobić instrukcjami cbw i cwde, możemy zrobić instrukcją movsx;

```
movsx   ax,al.    ;cbw
movsx   eax,ax    ;cwde
movsx   eax,al.   ;cbw następujące po cwde
```

Jednak, instrukcje cbw i cwde są krótsze i czasami szybsze. Ta instrukcja jest dostępna tylko na procesorach 80386 i późniejszych. zauważmy też, że nie ma bezpośredniego ekwiwalentu movsx dla instrukcji cwd i cdq.

Instrukcja movzx działa podobnie jak movsx, z wyjątkiem tego, że rozszerza wartości bez znaku przez powielanie zer zamiast wartości ze znakiem przez powielanie znaku. Składnia jest taka sam jak dla movsx, z wyjątkiem oczywiście użycia mnemonika movzx zamiast movsx.

Jeśli chcemy powielić zero ośmiobitowej wartości do 16 bitów (np. al. do ax) prosta instrukcja mov jest szybsza i krótsza niż movzx. Na przykład,

```
mov bh,0
```

jest krótsze niż

```
movzx bx, bl
```

Oczywiście, jeśli przenosimy dane do różnych 16 bitowych rejestrów (np. movzx bx ,al.) instrukcja movzx jest lepsza.

Podobnie jak instrukcja movsx instrukcja movzx jest dostępna tylko na procesorach 80386 lub późniejszych. Powielanie zer lub znaków nie wpływa na żadną z flag.

6.4.2 INSTRUKCJA BSWAP

Instrukcja bswap dostępna jest tylko na 80486 (tak,486) i późniejszych procesorach, konwertuje pomiędzy 32 bitowymi wartościami little endian i big endian**.** Ta instrukcja akceptuje tylko pojedynczy 32 bitowy operand rejestr. Wymienia pierwszy bajt z czwartym a drugi bajt z trzecim. Składnia dla instrukcji to;

bswap reg $_{32}$

Gdzie reg_{32} jest 32 bitowym rejestrem ogólnego przeznaczenia.

Procesory z rodziny Intela używają organizacji pamięci znanej jako **little endian byte organization (LEBO)** W LEBO ,najmniej znaczący bajt wielobajtowej sekwencji staje się najniższym adresem w pamięci. Na przykład ,bity zero do siedem 32 bitowej wartości pojawiają się jako drugi adres w pamięci; bity 16 do 23 pojawiają się w trzecim bajcie, a bity 24 do 31 w czwartym bajcie.

Inną popularną organizacją pamięci jest **big endian.** W schemacie big endian bity 24 do 31 pojawiają się w pierwszym (najniższym adresie),bity 16 do 23 w drugim bajcie, bity 8 do 15 w trzecim bajcie a bity o do 7 w czwartym bajcie. CPU takie jak rodzina Motoroli 68000 używane przez Apple w Macintoshach i wielu chipach RISC, stosuje schemat big endian.

Normalnie nie musimy martwić się o organizację bajtów w pamięci, ponieważ pisane programy dla procesorów Intela w asemblerze, nie pracują na procesorach 68000.Jednak,jest dosyć powszechna wymiana danych między maszynami z różnymi organizacjami bajtów. Niestety, 16 i 32 bitowe wartości w maszynie big endian nie tworzą prawidłowych wyników kiedy używamy ich na maszynach little endian. Tam wchodzi instrukcja bswap. Pozwala ona nam łatwo skonwertować 32 bitową wartość big endian do 32 bitowej wartości little endian.

Jednym interesującym zastosowaniem instrukcji bswap jest uzyskanie dostępu do drugiego zbioru rejestrów ogólnego przeznaczenia.. Jeśli używamy 16 bitowych rejestrów ogólnego przeznaczenia w naszym kodzie, możemy zdublować liczbę dostępnych rejestrów poprzez użycie instrukcji bswap do wymiany danych z 16 bitowego rejestru z bardziej znaczącym słowem rejestru 32 bitowego. Na przykład, możemy trzymać dwie 16 bitowe wartości w eax i przenieść odpowiednią wartość do ax jak następuje:

```
< Jakieś obliczenia, które zostawiają wynik w AX >
        bswap   eax
< Jakieś dodatkowe obliczenia  wymagające AX >
        bswap   eax
< Jakieś obliczenia wymagające oryginalnej wartości AX >
        bswap   eax
< Obliczenia wymagające drugiej kopii AX >
```

Możemy użyć tej techniki w 80486 dla uzyskania dwóch kopii ax, bx, cx ,dx ,si, di i bp. Musimy być bardzo ostrożni jeśli używamy tej techniki z rejestrem sp.

Notka :przy konwertowaniu 16 bitowej wartości big endian do 16 bitowej wartości little endian używamy instrukcji 80x86 xchg. Na przykład, jeśli ax zawiera 16 bitową wartość big endian możemy zamienić ją na 16 bitową wartość little endian (lub vice versa) używając:

```
        xchg    al., ah
```

Instrukcja bswap nie wpływa na żadną z flag w rejestrze flag 80x86

6.4.3 INSTRUKCJA XLAT

Instrukcja xlat przenosi wartość do rejestru al. w oparciu o tablicę połączeń w pamięci. Robi to następująco:

```
        temp := al+bx
        al=ds.:[temp]
```

to znaczy bx wskazuje tablicę w bieżącym segmencie danych. Xlat zastępuje wartość w al bajtem spod offsetu oryginalnego w al. Jeśli al zawiera cztery, xlat zastępuje wartość w al piątą pozycją (offset cztery) w środku tablicy wskazywanej przez ds:bx. Instrukcja xlat przybiera formę:

```
        xlat
```

Zazwyczaj nie ma operandu. możemy wyszczególnić jeden ale asembler praktycznie go zignoruje. Jedynym celem wyszczególnienia operandu jest to ,że możemy dostarczyć nadpisania przedrostka segmentu

```
        xlat    esL Table
```

Powie to asemblerowi aby wyemitował bajt es:przedrostek segmentu przed instrukcją. musimy jeszcze załadować bx adresem Table; powyższa forma nie dostarcza adresu Table do instrukcji. Tylko przedrostek przesłonięcia segmentu w operandzie jest znaczący.

Instrukcja xlat nie wpływa na rejestr flag 80x86.

6.5 INSTRUKCJE ARYTMETYCZNE

80x86 dostarcza wielu operacji arytmetycznych: dodawanie ,odejmowanie ,negacja, mnożenie ,dzielenie / modulo (reszta) i porównywanie dwóch wartości. Instrukcje wykonujące te operacje to add, adc, sub

,sbb, mul ,imul ,div, idiv, cmp, neg, inc, dec, xadd, cmpxchg i kilka różnych instrukcji konwersji: aaa, aad, aam, aas i das. Następne sekcje opisują te instrukcje szczegółowo.

Ogólne formy dla tych instrukcji to:

```
add     dest, src             dest := dest + src
adc     dest, src             dest := dest + src + C
SUB     dest, src             dest := dest - src
sbb     dest, src             dest := dest - src - C
mul     src                   acc := acc * src
imul    src                   acc := acc * src
imul    dest, src1, imm_src   dest := src1 * imm_src
imul    dest, imm_src         dest := dest * imm_src
imul    dest, src             dest := dest * src
div     src                   acc := xacc /-mod src
idiv    src                   acc := xacc /-mod src
cmp     dest, src             dest - src (and set flags)
neg     dest                  dest := - dest
inc     dest                  dest := dest + 1
dec     dest                  dest := dest - 1
xadd    dest, src             (see text)
cmpxchg operand1, operand2    (see text)
cmpxchg8ax, operand           (see text)
aaa                           (see text)
aad                           (see text)
aam                           (see text)
aas                           (see text)
daa                           (see text)
das                           (see text)
```

6.5.1 INSTRUKCJE DODAWANIA: ADD,ADC,INC.XADD,AAA I DAA

Instrukcje te przybierają formy:

```
add     reg, reg
add     reg, mem
add     mem, reg
add     reg, dana natychmiastowa
add     mem, dana natychmiastowa
add     eax/ax/al., dana natychmiastowa

formy adc są identyczne jak ADD

inc     reg
inc     mem
inc     reg16
xadd    mem, reg
xadd    reg, reg
aaa
daa
```

Zauważmy ,że instrukcje aaa i daa używają niejawnego trybu adresowania i nie zawierają operandów.

6.5.1.1 INSTRUKCJE ADD I ADC

Składnia add i adc (dodawanie z przeniesieniem) jest podobna do mov. Podobnie jak mov, są specjalne formy dla rejestrów ax /eax które są bardziej wydajne. W odróżnieniu od mov ,nie możemy dodać wartości do rejestru segmentowego tymi instrukcjami.

Instrukcja add dodaje zawartość operandu źródłowego do operandu przeznaczenia. Na przykład, add ax, bx, dodaje bx do ax zostawiając sumę w ax. Add oblicza dest := dest +source podczas gdy Adc oblicza dest := dest +source +C gdzie C przedstawia wartość flagi przeniesienia (carry).Dlatego też jeśli flaga przeniesienia jest wyczyszczona przed wykonaniem, adc zachowuje się dokładnie jak instrukcja add.

Obie instrukcje wpływają na flagi identycznie. Ustawiają flagi jak następuje:

* Flaga przepełnienia oznacza przepełnienie liczby ze znakiem
* Flaga przeniesienia oznacza przepełnienie liczby bez znaku

* Flaga znaku oznacza wynik ujemny (tj. najbardziej znaczący bit wyniku wynosi jeden)
* Flaga przeniesienia połówkowego zawiera jeden jeśli przepełnienie BCD występuje w mnij znaczącym nibblu
* Flaga parzystości jest ustawiona lub wyczyszczona w zależności od parzystości najmniej znaczących ośmiu bitów wyniku. Jeśli jest parzysta liczba bitów jeden w wyniku, instrukcja ADD ustawi flagę parzystości na jeden Jeśli jest nieparzysta liczba bitów w wyniku, instrukcja ADD wyzeruje flagę

Instrukcje add i adc nie wpływają na żadne inne flagi.

Instrukcje add i adc uznają ośmio- szesnasto i (w 80386 i późniejszych CPU) trzydziesto dwu bitowe operandy. Obydwa operandy źródłowy i przeznaczenia muszą być tego samego rozmiaru .Zobacz Rozdział Dziewiąty jeśli chcesz dodawać operandy których rozmiar jest różny.

Ponieważ nie ma dodawania pamięci do pamięci, musimy załadować operand pamięci do rejestru jeśli chcemy dodać dwie zmienne razem. Następujący przykładowy kod demonstruje możliwe formy dla instrukcji add:

```
; J:=K+M
        mov     ax, K
        add     ax, M.
        mov     J, ax
```

Jeśli chcemy dodać kilka wartości razem ,możemy łatwo obliczyć sumę w pojedynczym rejestrze:

```
: J:=K+M+N+P
        mov     ax, K
        add     ax, M
        add     ax, N
        add     ax, P
        mov     J, ax
```

Jeśli chcemy zredukować liczbę przypadków na procesorze 80486 lub Pentium możemy użyć kodu takiego jak ten:

```
        mov     bx, K
        mov     ax, M
        add     bx, N
        add     ax, P
        add     ax, bx
        mov     J, ax
```

Jedną rzeczą o której często zapominają początkujący programiści asemblerowi jest to, że możemy dodać rejestr do komórki pamięci. Czasami początkujący programiści nawet wierzą, że oba operandy muszą być w rejestrach ,kompletnie zapominając lekcje z Rozdziału Czwartego. 80x86 jest procesorem CISC, który pozwala nam używać trybów adresowania pamięci z różnymi instrukcjami jak add. Często jest bardziej wydajnie wykorzystać potencjał adresowania pamięci

```
; J:=K+J
        mov     ax, K
        add     J, ax
;Początkujący często kodują powyższe jako jedną z dwóch powyższych sekwencji
; To jest zbyteczne!
        mov     ax, J           ;rzeczywiście zły sposób obliczania
        mov     bx, K           ;J:=J+K
        add     ax, bx
        mov     J, ax

        mov     ax, J           ;lepiej, ale jeszcze nie dobry sposób
        add     ax, K           ;obliczania J :=J+K
        mov     J, ax
```

Oczywiście jeśli chcemy dodać stałą do komórki pamięci, potrzebujemy pojedynczej instrukcji.80x86 pozwala nam bezpośrednio dodać stałą do pamięci:

```
; J := J+2
        add     J, 2
```

Są specjalne formy instrukcji add i adc ,które dodają bezpośrednio stałe do rejestru al., ax lub eax. Formy te są krótsze niż standardowa instrukcja add reg, dana bezpośrednia. Inne instrukcje również dostarczają

krótszych form, kiedy używają tych rejestrów; dlatego też, powinniśmy utrzymywać obliczenia w rejestrze akumulatora (al., ax i eax) tak długo jak możliwe.

```
add     bl, 2           ;długa na trzy bajty
add     al., 2          ;długa na dwa bajty
add     bx, 2           ;długa na cztery bajty
add     ax, 2           ;długa na trzy bajty
itd.
```

Inne sprawy związane z używaniem małych znakowych stałych z instrukcjami add i adc. Jeśli wartość jest z zakresu –128..127, instrukcje add i adc powielają znak ośmiobitowej stałej natychmiastowej do koniecznego rozmiaru operandu przeznaczenia (osiem, szesnaście lub trzydzieści dwa bity)Dlatego powinniśmy próbować używać małych stałych, jeśli to możliwe, z instrukcjami add i adc

---

6.5.1.2 INSTRUKCJA INC

Instrukcja INC (increment – zwiększenie) dodaje jeden do własnego operandu.Za wyjątkiem flagi przeniesienia, inc ustawia flagi w ten sam sposób jak add operand, 1.

Zauważmy, że są dwie formy inc dla 16 i 32 bitowego rejestru. Są to instrukcje inc reg i inc reg $_{16}$.Instrukcje inc reg i inc mem są takie same. Ta instrukcja składa się z opcodu bajtu określonego przez bajt mod-reg-r/m. (zobacz Appendix D po szczegóły).instrukcja inc reg$_{16}$ ma pojedynczy opcod bajtu. Dlatego jest krótsza i zazwyczaj szybsza.

Operand inc może być ośmio- szesnasto- lub trzydziesto dwu bitowym rejestrem lub komórką pamięci..

Instrukcja inc jest często szybsza niż odpowiadająca jej instrukcja add reg, 1 lub add mem, 1.Faktycznie,instrukcja inc reg$_{16}$ jest długa na jeden bajt, więc okazuje się, że dwie takie instrukcje są krótsze niż porównywalne instrukcje add reg, 1;jednak dwie instrukcje zwiększania będą pracowały wolniej na bardziej nowoczesnych członkach rodziny 80x86.

Instrukcja inc jest bardzo ważna ponieważ dodawanie jeden do rejestru jest bardzo powszechną operacją. Zwiększanie zmiennych sterowania pętlą lub indeksowanie wewnątrz tablicy jest bardzo popularną operacją, doskonałą dla instrukcji inc .Fakt, że inc nie wpływa na flagę przeniesienia jest bardzo ważny. Pozwala to nam zwiększać indeksy tablicy bez wpływania na wynik operacji arytmetycznych o zwielokrotnionej precyzji (zobacz „Arytmetyczne i Logiczne Operacje” po więcej szczegółów o arytmetyce o zwielokrotnionej precyzji)

6.5.1.3 INSTRUKCJA XADD

Xadd (wymian i dodawanie) jest inną instrukcją 80486 (i późniejszych) procesorów. Nie pojawiła się w 80386 i wcześniejszych procesorach, instrukcja ta dodaje operand źródłowy do operandu przeznaczenia a sumę przechowuje w operandzie przeznaczenia. Jednak przed przechowaniem sumy kopiuje oryginalną wartość operandu przeznaczenia w operandzie źródłowym. następujący algorytm opisuję tę operację;

```
xadd    przez , źródło

temp :=przez
przez:= przez+ źródło
źródło:= temp
```

Xadd ustawia flagi tak jak instrukcja add. Instrukcja xadd pozwala na ośmio- szesnasto- i trzydziesto dwu bitowe operandy. Oba operandy, źródłowy i przeznaczenia muszą być tego samego rozmiaru.

6.5.1.4 INSTRUKCJE AAA I DAA

Instrukcje AAA (Modyfikowanie ASCII po dodawaniu) i DAA (Modyfikowanie dziesiętne dla dodawania) wspierają arytmetykę BCD. Poza tym rozdziałem ten tekst nie stosuje arytmetyki BCD lub ASCII ponieważ jest stosowana głównie dla sterowników aplikacji a nie ogólnego zastosowania programowania aplikacji. Wartości BCD są dziesiętnymi wartościami całkowitymi kodowanymi binarnie z jedną cyfrą dziesiętną na nibble’a. Wartość ASCII (numeryczna) zawiera pojedynczą cyfrę dziesiętną na bajt, bardziej znaczący nibble bajtu powinien zawierać zero.

Instrukcje aaa i daa modyfikują wynik binarnego dodawania do właściwego dla arytmetyki ASCII lub dziesiętnej. Na przykład, dodanie dwóch wartości BCD, dodamy je jak gdyby były wartościami binarnymi a potem wykonamy instrukcję daa w celu korekcji otrzymanego wyniku. Podobnie, możemy użyć instrukcji aaa dla modyfikacji wyniku dodawania ASCII wykonaniu instrukcji add .Proszę zapamiętać, że te dwie instrukcje zakładają, że operandy dodawania były właściwymi wartościami dziesiętnymi lub ASCII. Jeśli dodamy binarnie (nie- dziesiętnie lub nie- ASCII) wartości razem i spróbujemy zmodyfikować je tymi instrukcjami, nie otrzymamy poprawnego wyniku.

Wybór nazwy „arytmetyka ASCII” jest niefortunny, ponieważ te wartości nie są prawdziwymi znakami ASCII. Nazwa taka jak „nie upakowane BCD” byłaby bardziej odpowiednia. Jednak Intel używa nazwy ASCII,

więc ten tekst też będzie to robił aby uniknąć nieporozumień. Jeśli będziemy słyszeli termin „nie upakowane BCD" będzie chodziło o ten typ danych.

Aaa (która zazwyczaj wykonuje się po instrukcjach add, adc lub xadd) sprawdza wartość w al. dla przepełnienia BCD. Wykonuje to według takiego algorytmu:

```
if ( (al and 0Fh) > 9 or (AuxC[6] =1) ) then
                if (8088 or 8086)[6] then
                        al := al + 6
                else
                        ax := ax + 6
                endif
                ah := ah + 1
                AuxC := 1                       ;Set auxilliary carry
                Carry := 1                      ; and carry flags.
else
                AuxC := 0                       ;Clear auxilliary carry
                Carry := 0                      ; and carry flags.
endif
al := al and 0Fh
```

Instrukcja aaa jest użyteczna głównie dla dodawania łańcuchów cyfr gdzie jest dokładnie jedna cyfra dziesiętna na bajt w łańcuchu liczbowym. Ten tekst nie będzie się zajmował łańcuchami liczbowymi BCD i ASCII, więc możemy spokojnie zignorować te instrukcje teraz. Oczywiście, możemy użyć instrukcji aaa w każdej chwili jeśli musimy zastosować powyższy algorytm, ale będzie to raczej wyjątkowa sytuacja.

Instrukcja daa funkcjonuje podobnie jak aaa z wyjątkiem tego, że operuje wartościami upakowanego BCD zamiast jedną cyfrą na bajt nie upakowanych wartości stosowanych przez aaa. Podobnie jak aaa, głównym celem daa jest dodawania łańcuchów cyfr BCD (z dwoma cyframi na bajt) Algorytm dla daa:

```
if ( (AL and 0Fh) > 9 or (AuxC = 1)) then
                al := al + 6
                AuxC := 1                       ;Set Auxilliary carry.
endif
if ( (al > 9Fh) or (Carry = 1)) then
                al := al + 60h
                Carry := 1;                     ;Set carry flag.
endif
```

6.5.2 INSTRUKCJE ODEJMOWANIA: SUB,SBB,DEC,AAS I DAS

Instrukcje sub (odjąć),sbb (odjąć z pożyczką),dec (zmniejszanie),aas (modyfikowanie ASCII dla odejmowania) i das (modyfikowanie dziesiętne dla odejmowania) działają tak jak tego oczekujemy. Ich składnia jest bardzo podobna do tej z instrukcji add:

```
sub     reg, reg
sub     reg, mem
sub     mem, reg
sub     reg, dana bezpośrednia
sub     mem, dana bezpośrednia
sub     eax, ax, al., dana bezpośrednia
forma sbb jest identyczna jak sub
dec     reg
dec     mem
dec     reg16
aas
das
```

Instrukcja sub oblicza wartość przez := przez – źródło. Instrukcja sbb oblicza przez := przez – źródło – C. Zauważmy, że odejmowanie nie jest przemienne. Jeśli chcemy obliczyć wynik dla przez := źródło – przez musimy użyć kilku instrukcji ,zakładając, że musimy zachować operand źródłowy).

Jednym z tematów wartym omówienia jest to jak instrukcja sub wpływa na rejestr flag 80x86.Instrukcje sub, sbb i dec wpływają na flagi jak następuje:

- Ustawiają flagę zera jeśli wynik jest zero. Występuje to jeśli operandy są różne dla sub i sbb. Instrukcja dec ustawia flagę zera tylko kiedy zmniejsza wartość jeden
- Instrukcje te ustawiają flagę znaku jeśli wynik jest ujemny
- Ustawiają flagę przepełnienia jeśli wystąpi przepełnienie /niedomiar ze znakiem
- Ustawiają flagę przeniesienia połówkowego jeśli to konieczne dla arytmetyki BCD/ASCII
- Ustawiają flagę parzystości według liczby bitów jeden występujących w wartości wyniku
- Instrukcje sub i sbb ustawiają flagę przeniesienia jeśli wystąpi przepełnienie bez znaku. Zauważmy ,że instrukcja dec nie wpływa na flagę przeniesienia

Instrukcja aas, podobnie jak jej odpowiednik aaa, pozwala nam działać na łańcuchach liczb ASCII z jedną cyfrą dziesiętną (z zakresu 0..9) na bajt. Będziemy używali tej instrukcji po instrukcji sub lub sbb na wartości ASCII. Instrukcja ta używa następującego algorytmu:

```
if ( (al and 0Fh) > 9 or AuxC = 1) then
                al := al - 6
                ah := ah - 1
                AuxC := 1               ;Set auxilliary carry
                Carry := 1              ; and carry flags.
else
                AuxC := 0               ;Clear Auxilliary carry
                Carry := 0              ; and carry flags.
endif
al := al and 0Fh
```

Instrukcja das wykonuje te same operacje dla wartości BCD gdy używa algorytmu:

```
if ( (al and 0Fh) > 9 or (AuxC = 1)) then
                al := al -6
                AuxC = 1
endif
if (al > 9Fh or Carry = 1) then
                al := al - 60h
                Carry := 1              ;Set the Carry flag.
endif
```

Ponieważ odejmowanie nie jest przemienne nie możemy używać instrukcji sub tak swobodnie jak instrukcji add. Poniższy przykład demonstruje na jakie problemy możemy się natknąć:

```
; J := K - J
                mov     ax, K           ;This is a nice try, but it computes
                sub     J, ax           ; J := J - K, subtraction isn't
                                        ; commutative!

                mov     ax, K           ;Correct solution.
                sub     ax, J
                mov     J, ax
; J := J - (K + M) -- Don't forget this is equivalent to J := J - K - M
                mov     ax, K           ;Computes AX := K + M
                add     ax, M
                sub     J, ax           ;Computes J := J - (K + M)
                mov     ax, J           ;Another solution, though less
                sub     ax, K           ;Efficient
                sub     ax, M
                mov     J, ax
```

Zauważmy, że instrukcje sub i sbb podobnie jak add i adc dostarczają krótkich form do odejmowania stałych z rejestru akumulatora (al., ax lub eax).Z tej przyczyny powinniśmy próbować trzymać operacje arytmetyczne w rejestrze akumulatora tak długo jak to możliwe .Instrukcje sub i sbb dostarczają również krótszych form kiedy odejmujemy stałe z zakresu –128..+127 z komórki pamięci lub rejestru. Instrukcje te automatycznie powielają znak ośmiobitowej wartości ze znakiem do koniecznego rozmiaru przed wykonaniem odejmowania. Zajrzyj do Appendix D po szczegóły.

W praktyce ,nie potrzeba instrukcji które odejmują stałe z rejestru lub komórki pamięci – dodanie wartości ujemnej da ten sam wynik. Niemniej jednak Intel dostarczył instrukcji bezpośrednich.

Po wykonaniu instrukcji sub, bity kodów błędu (przeniesienia, znaku, przepełnienia i zera) w rejestrze flag zawierają wartości które możemy przetestować aby zobaczyć czy jeden z operandów sub jest równy, nie równy, mniejszy niż, mniejszy niż lub równy, większy niż lub większy niż lub równy dla inne operacji. Zobacz instrukcję cmp po więcej szczegółów.

6.5.3 INSTRUKCJA CMP

Instrukcja cmp (porównanie) jest podobna do instrukcji sub z jednym zasadniczym wyjątkiem – nie przechowuje różnicy w operandzie przeznaczenia. Składnia dla instrukcji cmp jest bardzo podobna do sub, ogólna forma

```
cmp     przez, źródło
```

Określone formy:

```
cmp     reg, reg
cmp     reg, mem
cmp     mem, reg
cmp     reg, dana bezpośrednia
cmp     mem, dana bezpośrednia
cmp     eax/ax/al., dana bezpośrednia
```

Instrukcja cmp uaktualnia flagi 80x86 według wyników operacji odejmowania (przez – źródło).Możemy przetestować wynik porównania poprzez sprawdzenie właściwych flag w rejestrze flag. Po szczegóły na temat jak to się robi, zajrzyj do „Ustawienia Instrukcji Warunkowych" i „Instrukcje skoków warunkowych".

Zazwyczaj chcielibyśmy wykonać instrukcję skoku warunkowego po instrukcji cmp. Te dwa kroki procesu, porównanie dwóch wartości i ustawienie bitów flag, potem testowanie bitów flag przez instrukcje skoków warunkowych jest bardzo wydajnym mechanizmem dla podejmowani decyzji przez program.

Prawdopodobnie pierwszą rzeczą od jakiej zaczniemy badanie instrukcji cmp jest przyjrzenie się jak instrukcja cmp wpływa na flagi. Rozpatrzmy następującą instrukcje cmp:

```
cmp     ax, bx
```

Instrukcja ta dokonuje obliczenia ax – bx i ustawia flagi w zależności od wyniku obliczenia. Flagi są ustawione jak następuje:

Z: flaga zera jest ustawiona jeśli i tylko jeśli ax = bx. jest to jedyny mement kiedy ax – bx tworzy w wyniku zero. W związku z tym, możemy użyć flagi zero to sprawdzenia równości bądź nierówności.

S: flaga znaku jest ustawiona na jeden jeśli wynik jest ujemny. Na pierwszy rzut oka, możemy pomyśleć, że flaga będzie ustawiona jeśli ax jest mniejsze niż bx, ale nie jest tak zwykle. Jeśli ax = 7FFFh a bx =-1 (0FFFFh) odjęcie ax od bx da nam 8000h,które jest ujemne (więc flaga znaku będzie ustawiona)Więc, dla porównania liczb całkowitych ze znakiem flaga znaku nie zawiera właściwego stanu. Dla operandu bez znakowego, rozważmy ax=0FFFFh i bx =1.Ax jest większe niż bx ale ich różnica wynosi 0FFFEh czyli jest jeszcze negatywna. Okazuje się, że flaga znaku i flaga przepełnienia, wzięte razem mogą być używane dla porównania dwóch wartości ze znakiem.

O: flaga przepełnienia jest ustawiana po operacji cmp jeśli różnica między ax i bx tworzy przepełnienie lub niedomiar. Jak wspomniano powyżej, flaga znaku i flaga przepełnienia są używane kiedy wykonujemy porównanie ze znakiem.

C: flaga przeniesienia jest ustawiana po operacji cmp jeśli odejmowanie bx od ax wymaga pożyczki. Zdarza się to tylko wtedy kiedy ax jest mniejsze niż bx gdzie ax i bx są wartościami bez znaku.

Instrukcja cmp również wpływa na flagi parzystości i przeniesienia połówkowego, ale rzadko będziemy testować te dwie flagi po operacji porównania. Dajmy na to, że instrukcja cmp ustawi flagi w ten sposób, możemy spróbować porównać dwa operandy z następującymi flagami:

```
cmp     Operand1, Operand2
```

Unsigned operands:	Signed operands:
Z: equality/inequality	Z: equality/inequality
C: Oprnd1 < Oprnd2 (C=1) Oprnd1 >= Oprnd2 (C=0)	C: no meaning
S: no meaning	S: see below
O: no meaning	O: see below

For signed comparisons, the S (sign) and O (overflow) flags, taken together, have the following meaning:
If ((S=0) and (O=1)) or ((S=1) and (O=0)) then Oprnd1 < Oprnd2 when using a signed comparison.
If ((S=0) and (O=0)) or ((S=1) and (O=1)) then Oprnd1 >= Oprnd2 when using a signed comparison.

Tablica 27: Ustawienia wskaźników po CMP

Aby zrozumieć dlaczego te flagi są ustawione w ten sposób, rozważmy następujący przykład:

Oprnd1	minus	Oprnd2	S	O
0FFFF (-1)	-	0FFFE (-2)	0	0
08000	-	00001	0	1
0FFFE (-2)	-	0FFFF (-1)	1	0
07FFF (32767)	-	0FFFF (-1)	1	1

Pamiętajmy, że operacja cmp jest w rzeczywistości odejmowaniem, dlatego też, pierwszy przykład powyżej oblicza (-1)-(-2) co daje (+1).wynik jest dodatni a przepełnienie nie występuje więc flagi S i O mają zero. Ponieważ (S xor O) wynosi zero,$Operand_1$ jest większy niż lub równy $Operandowi_2$.

W drugim przykładzie, instrukcja cmp będzie obliczała (-32768) – (+1) co daje (-32769).Ponieważ 16 bitowa wartość całkowita ze znakiem nie może przedstawić tej wartości, wartość zawija się do 7FFFh (+32767) i ustawia flagę przepełnienia. Ponieważ wynik jest dodatni (przynajmniej zawartości 16 bitów) flaga znaku jest wyzerowana. Ponieważ (S xor O) wynosi jeden,$Operand_1$ jest mniejszy niż $Operand_2$.

W trzecim przykładzie, cmp oblicza (-2)-(-1) czyli mamy (-1).Nie występuje żadne przepełnienie więc flaga O wynosi zero, wynik jest ujemny więc flaga znaku ma wartość jeden. Ponieważ (S xor O) to jeden,$Operand_1$ jest mniejszy niż $Operand_2$.

W czwartym (i końcowym) przykładzie, cmp oblicza (+32767) –(-1).Tworzy to (+32768),ustawia flagę przepełnienia. Co więcej wartość zawija się do 8000h (-32768) więc flaga znaku jest również ustawiona. Ponieważ (xor O) wynosi zero,$Operand_1$ jest większy niż lub równy $Operandowi_2$.

6.5.4 INSTRUKCJE CMPXCHG I CMPXCHG8B

Instrukcja cmpxchg (porównanie i wymiana) jest dostępna tylko na 80486 i późniejszych procesorach. Ich składnia:

```
cmpxchg     reg, reg
cmpxchg     mem, reg
```

Operandy muszą być tego samego rozmiaru (osiem, szesnaście lub trzydzieści dwa bity).Ta instrukcja również używa rejestru akumulatora: automatycznie wybiera al ,ax lub eax dopasowując do rozmiaru operandów.

Ta instrukcja porównuje al, ax lub eax z pierwszym operandem i ustawia flagę zera jeśli są równe. Jeśli tak, wtedy cmpxchg kopiuje drugi operand do pierwszego. Jeśli nie są równe, cmpxchg kopiuje pierwszy operand do akumulatora. Następujący algorytm opisuje tą operację:

```
cmpxchg             operand1, operand2
if ({al/ax/eax} = operand1) then[8]
        zero := 1                           ;Set the zero flag
        operand1 := operand2
else
        zero := 0                           ;Clear the zero flag
        {al/ax/eax} := operand1
endif
```

Cmpxchg wspiera pewne struktury danych systemu operacyjnego wymagające operacji atomowych (są to operacje, których system nie może przerwać) i semafory .Oczywiście, jeśli możemy wstawić powyższy algorytm do naszego kodu, możemy użyć instrukcji cmpxchg jako właściwą.

Notka: w odróżnieniu od instrukcji cmp, instrukcja cmpxchg wpływa tylko na flagę zera 80x86.Nie możemy testować flag po cmpxchg podobnie jak po instrukcji cmp.

Procesor Pentium wspiera 64 bitową instrukcję porównania i wymiany – cmpxchg8b.Jej składnia:

cmpxchg8b ax, mem_{64}

Instrukcja ta porównuje 64 bitową wartość w edx:eax z wartością pamięci. Jeśli są równe, Pentium przenosi ecx:ebx do komórki pamięci, w przeciwnym razie ładuje edx:eax z komórki pamięci .Instrukcja ta ustawia flagę zera według wyniku. nie wpływa na żadne inne flagi.

6.5.5 INSTRUKCJA NEG

Instrukcja NEG (negacja) stosuje uzupełnia do dwóch bajtu lub słowa. bierze pojedynczą (przeznaczenie) operację i neguje ją .Składnia dla tej instrukcji:

```
neg     przeznaczenie
```

Wykonuje następujące obliczenie:

```
dest := 0 –dest
```

To skutecznie odwraca znak operandu przeznaczenia.

Jeśli operand to zero, jego znak nie zmienia się ,chociaż czyści flagę przeniesienia. Negowanie każdej innej wartości ustawia flagę przeniesienia. Negowanie bajtu zawierającego –128,słowa zawierającego –32768 lub podwójnego słowa zawierającego –2,147,483,648 nie zmienia operandu, ale ustawi flagę przepełnienia. NEG zawsze uaktualnia flagi A,S,P i Z podobnie kiedy używaliśmy instrukcji sub.

Dostępne postacie to;

```
neg     reg
neg     mem
```

Operandy mogą być ośmio ,szesnasto lub (na 80386 i późniejszych) trzydziesto dwu bitowe.
Kilka przykładów:

```
; J := -J
        neg     J
;J := -K
        mov     ax, K
        neg     ax
        mov     J, ax
```

6.5.6 INSTRUKCJE MNOŻENIA: MUL,IMUL I AAM

Instrukcje mnożenie dostarczają nam możliwość poczucia nieregularności w zbiorze instrukcji 80x86.instrukcje takie jak add, adc, sub i wiele innych w zbiorze instrukcji 80x86 używa bajtu mod-reg-r/m. dla wsparcia dwóch operandów. Niestety, nie ma dość bitów w bajtach opcodach 80x86 aby wesprzeć wszystkie instrukcje, więc 80x86 używa bitów reg w bajcie mod-reg-r/m. jako rozszerzenia opcodu. Na przykład inc ,dec i neg nie wymagają dwóch operandów, więc CPU 80x86 używają bitów reg jako rozszerzenia do ośmiu bitów opcodu.. Pracuje to świetnie dla pojedynczych operandów instrukcji, pozwalając projektantom Intela kodować kilka instrukcji (w rzeczywistości, osiem) z pojedynczym opcodem.

Niestety instrukcje mnożenia wymagają specjalnego traktowania a projektanci Intela nadal pragnęli skracać opcody więc zaprojektowali instrukcje mnożenia do używania z pojedynczym operandem. Pole reg zawiera rozszerzony opcod zamiast wartości rejestru .Oczywiście mnożenie jest funkcją dwóch operandów. Ta nieregularność czyni stosowanie mnożenia w 80x86 trochę bardziej trudniejszym niż innych instrukcji ponieważ jeden operand musi być w rejestrze akumulatora .Intel zaadoptował to nie ortogonalne podejście ponieważ uznali ,że programiści będą używali mnożenia w dużo mniejszym stopniu niż instrukcji takich jak add czy sub.

Jedynym problemem z dostarczeniem tylko formy mod-reg-r/m. instrukcji jest to, że nie możemy pomnożyć rejestru akumulatora przez stałą; bajt mod-reg-r/m. nie wspiera bezpośredniego trybu adresowania .Intel szybko odkrył, że musi wesprzeć mnożeni przez stałą i zmienił to w procesorze 80286.Było to szczególnie ważne przy dostępie do wielowymiarowych tablic. Zanim pojawił się 80386,Intel uogólnił jedną postać operacji mnożenia przeznaczoną dla standardowego operandu mod-reg-r/m.

Są dwie formy instrukcji mnożenia: mnożenie bez znaku (mul) i mnożenie ze znakiem (imul).W odróżnieniu od dodawania i odejmowania, musimy oddzielić instrukcje dla tych dwóch operacji..

Instrukcje mnożenia przyjmują następujące formy:

Mnożenie Bez Znaku;

```
mul     reg
mul     mem
```

Mnożenie Ze Znakiem (Całkowite):

```
imul    reg
imul    mem
imul    reg, reg,  dana bezpośrednia
imul    reg, mem, dana bezpośrednia
imul    reg, dana bezpośrednia
imul    reg, reg
imul    reg, mem
```

Operacja Mnożenia BCD:

```
aam
```

Jak możemy zobaczyć, instrukcje mnożenia są prawdziwie nieuporządkowane. Gorzej jeszcze ,musimy używać procesorów 80386 lub późniejszych aby otrzymać prawie pełną funkcjonalność. W końcu, jest kilka ograniczeń w tych instrukcjach, nie tak oczywistych powyżej. Niestety, jedyny sposób wykorzystania tych instrukcji to wprowadzenie tych operacji do pamięci.

Mul dostępna na wszystkich procesorach, mnoży bez znakowe 8- 16 lub 32 bitowe operandy .Zauważmy, że kiedy mnożymy dwie n-bitowe wartość ,wynik może wymagać 2*n bitów. Dlatego też, jeśli operand jest ośmio bitową wielkością ,wynik będzie wymagał szesnaście bitów. Podobnie, operand 16 bitowy tworzy 32 bitowy rezultat a 32 bitowy operand wymaga 64 bitów jako wyniku.

Instrukcja mul ,z ośmio bitowym operandem, mnoży rejestr al., przez operand i przechowuje 16 bitowy wynik w ax. Więc

```
mul     operand8
```

lub

```
imul    operand8
```

oblicza:

```
ax := al.* operand8
```

„*" przedstawia mnożenie bez znaku dla mul i mnożenie ze znakiem dla imul.

Jeśli wyszczególnimy 16 bitowy operand, wtedy mul i imul obliczają:

```
dx:ax := ax* operand16
```

„*" ma takie samo znaczenie jak powyżej a dx:ax oznacza, że dx zawiera bardziej znaczące słowo 32 bitowego wyniku a ax zawiera mniej znaczące słowo 32 bitowego wyniku.

Jeśli wyszczególnimy 32 bitowy operand, wtedy mul i imul obliczają co następuje:

```
Edx:eax := eax * operand32
```

„*" ma takie samo znaczenie jak powyżej a edx:eax oznacza, że edx zawiera bardziej znaczące podwójne słowo z 64 bitowego wyniku a eax zawiera mniej znaczące podwójne słowo z 64 bitowego wyniku

Jeśli iloczyn 8x8,16x16 lub 32x32 bitów wymaga więcej niż, (odpowiednio) osiem, szesnaście lub trzydzieści dwa bity, instrukcje mul i imul ustawiają flagi przeniesienia i przepełnienia.

Mul i imul pozmieniają flagi A,P,S i Z. Zwłaszcza zauważmy, że flagi znaku i zera nie zawierają znaczących wartości po wykonaniu tych dwóch instrukcji.

Imul (mnożenie całkowite) działa na operandach ze znakiem. Jest wiele różnych form tej instrukcji ponieważ Intel próbował uogólnić tą instrukcję w kolejnych procesorach. Poprzedni paragraf omawiał pierwszą formę instrukcji imul, z pojedynczym operandem. następne trzy formy instrukcji imul są dostępne tylko na procesorach 80286 i późniejszych. dostarczają one zdolności do mnożenie rejestry przez wartość bezpośrednią. Ostatnie dwie formy, dostępne tylko na 80386 i późniejszych procesorach, dostarczają zdolności mnożenia przypadkowego rejestru przez inny rejestr lub komórkę pamięci.

```
imul    operand1, operand2, immediate

imul    reg16, reg16, immediate8
imul    reg16, reg16, immediate16
imul    reg16, mem16, immediate8
imul    reg16, mem16, immediate16
imul    reg16, immediate8
imul    reg16, immediate16
imul    reg32, reg32, immediate8
imul    reg32, reg32, immediate32
imul    reg32, mem32, immediate8
imul    reg32, mem32, immediate32
imul    reg32, immediate8
imul    reg32, immediate32
```

Instrukcje imul reg, dana bezpośrednia jest specjalną składnią dostarczoną przez asembler. Kodowanie dla tych instrukcji jest takie same jak imul reg, reg, dana bezpośrednio. Asembler po prostu dostarcza taką samą wartość rejestru dla obu operandów.

Instrukcje te obliczają:

operand$_1$:= operand$_2$ * dana bezpośrednia

operand$_1$:= operand$_1$ * dana bezpośrednia

Poza liczbą operandów, jest kilka różnic miedzy tymi formami a pojedynczym operandem instrukcji mul/ imul:

- Nie ma dostępnego mnożenia bitów 8x8 (operand$_8$ bezpośredni po prostu daje prostszą formę tej instrukcji. Wewnętrznie, CPU powiela znak operandu do 16 lub 32 bitów jeśli to konieczne).
- instrukcje te nie tworzą wyniku 2*n bitów. To znaczy, mnożenie 16x16 daje wynik 16 bitowy. podobnie mnożenie 32x32 daje 32 bitowy wynik. Instrukcje te ustawiają flagi przeniesienia przepełnienia jeśli wynik nie mieści się w rejestrze przeznaczenia.
- Wersja 80286 instrukcji imul pozwala na operand bezpośredni, standardowe instrukcje mul /imul nie.

Ostatnie dwie formy instrukcji imul są dostępne tylko na procesorach 80386 i późniejszych. Z tymi dodatkowymi formami, instrukcja imul jest prawie tak ogólna jak instrukcja add:

```
imul    reg, reg
imul    reg, mem
```

Instrukcje te obliczają

reg := reg * reg

i reg := reg * mem

Oba operandy muszą być tego samego rozmiaru. Dlatego też, podobnie jak forma dla 80286 instrukcji imul, musimy sprawdzić flagi przeniesienia i przepełnienia do wykrycia przepełnienia. Jeśli wystąpiło przepełnienie, CPU gubi bardziej znaczące bity wyniku.

Ważna Uwaga: Zapamiętajmy, że flaga zera zawiera nieokreślony wynik po wykonaniu instrukcji mnożenia. Nie możemy przetestować flagi zera aby zobaczyć czy wynik to zero po mnożeniu. Podobnie instrukcje te zmieniają flagę znaku. jeśli musimy sprawdzić te flagi ,porównamy wynik do zera po przetestowaniu flag przeniesienia i przepełnienia.

Instrukcja aam (Modyfikacja ASCII Po Mnożeniu),podobnie jak aaa i aas, pozwala nam modyfikować nie upakowane dziesiętnie wartości po mnożeniu. Instrukcja ta operuje bezpośrednio na rejestrze ax. Zakładając, że pomnożymy razem dwie wartości ośmiobitowe z zakresu 0..9 a wynik jest usytuowany w ax (w rzeczywistości wynik jest usytuowany w al, ponieważ 9*9 daje nam 81,największą możliwą wartość; ah musi zawierać zero).Instrukcja ta dzieli ax przez 10 i zostawia iloraz w ah a resztę w al:

ah := ax div 10

al. := ax mod 10

W odróżnieniu do innych instrukcji modyfikacji dziesiętnej/ ASCII, program asemblerowy regularnie używa aam ponieważ konwersja pomiędzy podstawami liczb używa tego algorytmu.

Notka: instrukcja aam składa się z dwóch bajtów opcodu, drugi z nich jest stałą bezpośrednią 10.Programiści asemblerowi odkryli, że jeśli zastąpi tą stałą inną wartością bezpośrednią, możemy zmienić dzielnik w powyższym algorytmie. Jest to jednak cecha nie udokumentowana. Działa ona na wszystkich odmianach procesorów Intela, tworząc dane ,ale nie ma gwarancji, że Intel będzie wspierał ją w następnych procesorach.Ocywiście80286 i późniejsze procesory pozwalają nam mnożyć przez stała, więc ta sztuczka jest prawie nie potrzebna w nowoczesnych systemach.

Nie ma instrukcji dam (modyfikowanie dziesiętne dla mnożenia) w procesorach 80x86

Być może najbardziej użytecznym zastosowaniem instrukcji imul jest obliczanie offsetów w tablicach wielowymiarowych. Rzeczywiście, jest to prawdopodobnie główny powód dla którego Intel dodał zdolność mnożenia rejestru przez stałą w procesorze 80286.W Rozdziale Czwartym, ten tekst używa standardowej instrukcji mul dla obliczania indeksów tablic. Jednakże, rozszerzona składnia instrukcji imul daje nam lepszy wybór jak pokazuje następujący przykład:

```
MyArray         word    8 dup ( 7 dup ( 6 dup (?)))     ;8x7x6 array.
J               word    ?
K               word    ?
M               word    ?
                 .
                 .
                 .
; MyArray [J, K, M]  := J + K - M
                mov     ax, J
                add     ax, K
                sub     ax, M

                mov     bx, J           ;Array index :=
                imul    bx, 7           ;           ((J*7 + K) * 6 + M) * 2
                add     bx, K
                imul    bx, 6
                add     bx, M
                add     bx, bx          ;BX := BX * 2

                mov     MyArray[bx], ax
```

Nie zapomnijmy, że instrukcje mnożenia są bardzo wolne; często bywają wolniejsze niż instrukcje dodawania. Są szybsze sposoby mnożenia wartości przez stałą. Zobacz :"Mnożenie bez MUL i IMUL" po więcej szczegółów.

6.5.7 INSTRUKCJE DZIELENIA: DIV,IDIV I ADD

Instrukcje dzielenia 80x86 wykonują dzielenie 64/32 (tylko 80386 i późniejsze),32/16 lub 16/8.Instrukcej te przyjmują formę:

```
div     reg             dla dzielenia bez znakowego
div     mem
idiv    reg             dla dzielenia ze znakiem
idiv    mem
aad                     modyfikowanie ASCII dla dzielenia
```

Instrukcja div wykonuje dzielenie bez znakowe. Jeśli operand jest ośmiobitowym operandem, div dzieli rejestr ax przez operand zostawiając iloraz w al. a reszta (modulo) w ah. Jeśli operand jest 16 bitową wielkością, wtedy instrukcja div dzieli 32 bitowy wielkość w dx:ax przez operand pozostawiając iloraz w ax a resztę w dx. Z 32 bitowym operandem (tylko 80386 lub późniejsze) div dzieli wartość 64 bitową w edx:eax przez operand pozostawiając iloraz w eax a resztę w edx.

W 80x86 nie możemy po prostu podzielić jednej ośmio bitowej wartości przez inną. Jeśli mianownik jest ośmio bitową wartością ,liczebnik musi być wartością szesnasto bitową. Jeśli musimy podzielić jedna ośmio bitową wartość bez znaku przez inną, musimy powielić zero liczebnika do szesnastu bitów. Możemy to osiągnąć poprzez załadowanie liczebnika do rejestru al a potem przesunąć zero do rejestru ah. Wtedy możemy podzielić ax przez operator mianownika uzyskując właściwy wynik. Opuszczenie powielenia zera dla al. przed wykonaniem div może spowodować w 80x86 uzyskanie niewłaściwego wyniku!

Kiedy musimy podzielić dwie szesnastobitowe wartości bez znaku, musimy powielić zero rejestru ax (który zawiera liczebnik) do rejestru dx. Właściwie ładujemy bezpośrednią wartość zero do rejestru dx .Jeśli musimy podzielić jedną 32 bitową wartość przez inną, musimy powielić zero rejestru eax do edx (poprzez załadowanie zer do edx) przed dzieleniem.

Jest jeszcze inna zasadzka instrukcji dzielenia 80x86:możemy popełnić fatalny błąd kiedy użyjemy tej instrukcji. Po pierwsze, możemy próbować dzielić wartość przez zero .Ponadto, iloraz może być zbyt długi do przechowania w rejestrze eax, ax lub al. Na przykład dzielenie 16/8 „8000h/2" tworzy iloraz 4000h z resztą zero. 4000h nie mieści się w ośmiu bitach. Jeśli zdarzy się coś takiego lub spróbujemy podzielić przez zero,80x86 wygeneruje przerwanie int 0.To zazwyczaj znaczy ,że BIOS wydrukuje „dzielenie przez zero: lub „błąd dzielenia" i przerwie wykonywanie programu. Jeśli zdarzy się to nam, prawdopodobnie nie powieliliśmy zera lub znaku naszego liczebnika przed wykonaniem operacji dzielenia. Ponieważ ten błąd przyczynia się do rozłożenia naszego programu, powinniśmy być bardzo ostrożni co do wartości które wybieramy kiedy używamy dzielenia.

Flagi przeniesienia połówkowego, przeniesienia, przepełnienia ,parzystości, znaku i zera są niezdefiniowane po operacji dzielenia. Jeśli wystąpi przepełnienie (lub spróbujemy dzielić przez zero) wtedy 80x86 wykona INT 0 (przerwanie 0).

Zauważmy, że 80286 i późniejsze procesory nie dostarczają specjalnych form dla idiv tak jak dla imul. Większość programów używa dzielenia mniej częściej niż używają mnożenia, więc projektanci Intela nie zawracali sobie głowy tworzeniem specjalnych instrukcji dla operacji dzielenia. Nie ma sposobu dzielenia przez wartość bezpośrednią. Musimy załadować wartość bezpośrednią do rejestru lub komórki pamięci i wykonać dzielenie przez ten rejestr lub komórkę pamięci.

Instrukcja aad (modyfikowanie ASCII przed dzieleniem) jest inną nie upakowaną operacją dziesiętną. Dzieli ona wartość BCD przed operacją dzielenia ASCII. Chociaż ten tekst nie stosuje arytmetyki BCD, instrukcja aad jest użyteczna dla innych operacji. Algorytm który opisuje tą instrukcję:

```
al := ah*10 + al
ah :=0
```

Instrukcja ta jest całkiem użyteczna dla konwertowania łańcuchów cyfr do wartości całkowitych (zobacz pytania na końcu tego rozdziału).

Następujący przykład pokazuje jak podzielić jedną 16 bitową wartość przez inną.

```
; J := K / M. (bez znaku)
        mov     ax, K           ;ustawienie dzielnej
        mov     dx, o           ; powielenie zera wartości bez znaku w ax do dx
        div     M
        mov     J, ax

; J := K/M. (ze znakiem)
```

```
                mov     ax, K           ;ustawienie dzielnej
                cwd                     ;powielenie znaku wartości ze znakiem w ax do dx
                idiv    M
                mov     J, ax

; J := (K*M.)/P
                mov     ax, K           ;Zauważmy, że instrukcja imul tworzy
                imul    M               ;32 bitowy wynik w DX:AX, więc nie
                idiv    P               ;musimy powielać znaku ax tutaj
                mov     J, ax           ; miejmy nadzieję, że wynik mieści się w 16 bitach
```

6.6 INSTRUKCJE LOGICZNE, OBROTU I BITOWE

Rodzina 80x86 dostarcza pięć logicznych instrukcji, cztery instrukcje obrotów i trzy instrukcje przesunięcia. Instrukcje logiczne to and, or, xor, test i not; obrotu to ror, rol, rcr i rcl; instrukcje przesunięcia to shl /sal ,shr i sar. Procesory 80386 i późniejsze dostarczają nawet bogatszy zbiór operacji. Są to bt, bts, btr, btc, bsf, bsr, shld, shrd i zbiór instrukcji warunkowych (setcc).

Instrukcje te mogą manipulować bitami, konwertują wartości, robią logiczne operacje, pakują i rozpakowują dane i robią operacje arytmetyczne. Ta sekcja omawia każdą z tych instrukcji szczegółowo.

6.6.1 INSTRUKCJE LOGICZNE: AND,OR,XOR I NOT

Logiczne instrukcje 80x86 działają na podstawie bit przez bit. Istnieją dwie ośmio, szesnasto i trzydziesto dwu bitowe wersje każdej instrukcji. Instrukcje and, not, or i xor robią co następuje:

```
and     przez, źródło       ;przez := przez and źródło
or      przez, źródło       ;przez := przez or źródło
xor     przez, źródło       ;przez := przez xor źródło
not     przez               ‘przez := not przez
```

Określone warianty to

```
and     reg, reg
and     mem, reg
and     reg, mem
and     reg, dana bezpośrednia
and     mem, dana bezpośrednia
and     eax/ax/al., dana bezpośrednia

or używa tych samych form jak AND
xor     używa tych samych form co AND

not     rejestr
not     pamięć
```

Za wyjątkiem, instrukcje not wpływają na flagi jak następuje:

- Czyszczą flagę przeniesienia
- Czyszczą flagę przepełnienia
- Ustawiają flagę zera jeśli wynik to zero, w przeciwnym razie czyszczą ją.
- Kopiują bardziej znaczący bit wyniku do flagi znaki.
- Ustawiają flagę parzystości według parzystości (liczby bitów jeden) w wyniku
- Zmieniają flagę przeniesienia połówkowego

Instrukcja not nie wpływa na żadną z flag.

Testowanie flagi zera jest szczególnie użyteczne. Instrukcja and ustawia flagę zera jeśli dwa operandy nie mają żadnej jedynki na odpowiadających sobie pozycjach bitów (ponieważ uzyskujemy wynik zero);na przykład, jeśli operand źródłowy zawierał pojedynczy jeden bit, wtedy flaga zera będzie ustawiona jeśli odpowiadające bit przeznaczenia wynosi zero, w innym razie będzie jedynką. Instrukcja or ustawia tylko flagę zera jeśli oba operandy zawierają zero. instrukcja xor ustawi flagę zera tylko jeśli oba operandy są różne. Zauważmy, że operacja xor stworzy wynik zero jeśli i tylko jeśli dwa operandy są równe. Wielu programistów powszechnie używa tego faktu do czyszczenia rejestru szesnasto bitowego do zera ponieważ instrukcja w postaci

```
xor     reg16, reg16
```

jest krótsza niż porównywalna instrukcja mov reg, 0.

Podobnie jak instrukcje dodawania i odejmowania, instrukcje and, or i xor dostarczają specjalnych form wymagających rejestru akumulatora i danej bezpośredniej. Te formy są krótsze i czasami szybsze niż ogólne formy „rejestr, dana bezpośrednia". Chociaż nikt normalnie nie myśli o działaniu na danych znakowych z tymi instrukcjami,80x86 dostarcza specjalnej formy instrukcji „reg /mem, dana bezpośrednia" która powiela znak w zakresie –128..127 do szesnastu lub 32 bitów, jeśli to konieczne.

Wszystkie operandy tych instrukcji muszą być tego samego rozmiaru. Na pre-80386 procesorach mogły być ośmio lub szesnasto bitowe. Na 80386 i późniejszych procesorach mogą być długości 32 bitów. Instrukcje te obliczają oczywiste operacje logiczne na poziomie bitowym na swoich operandach, zobacz Rozdział Jeden po więcej szczegółów o tych operacjach.

Możemy użyć instrukcji and do ustawienia wyselekcjonowanych bitów na zero w operandzie przeznaczenia. Jest to znane jako maskowanie danych. Podobnie ,możemy użyć instrukcji or do wymuszenia pewnych bitów na jeden w operandzie przeznaczenia; zobacz „Operacje maskowania operacją OR".

Możemy użyć tych instrukcji razem z instrukcjami przesunięcia i obrotu opisanymi dalej ,do pakowania i rozpakowywania danych. Zobacz „Pakowanie i Rozpakowywanie typów danych" po więcej szczegółów.

6.6.2 INSTRUKCJE PRZESUNIĘCIA: SHL/SAL,SHR,SAR,SHLD I SHRD

80x86 wspiera trzy różne instrukcje przesunięcia (shl i sal są tymi samymi instrukcjami): shl (przesunięcie w lewo),sal (arytmetyczne przesunięcie w lewo), shr (przesunięcie w prawo) i sar (przesunięcie arytmetyczne w prawo).Procesory 80386 i późniejsze dostarczają dwie dodatkowe przesunięcia: shld i shrd.

Instrukcje przesunięcia przenoszą bity w rejestrze lub komórce pamięci. Ogólny format dla instrukcji przesunięcia to

```
shl     przez, liczba
sal     przez, liczba
shr     przez, liczba
sar     przez, liczba
```

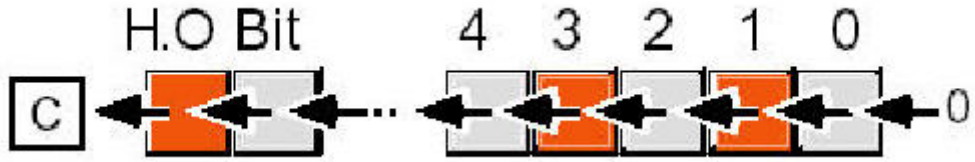
Przez jest wartością do przesunięcia a liczba wyszczególnia liczbę o ile bitów chcemy przesunąć. Na przykład, instrukcja shl przesuwa bity w operandzie przeznaczenia w lewo o liczbę bitów wyszczególnioną w operandzie liczba. Instrukcje shld i shrd używają formatu;

```
shld    przez ,źródło, liczba
shrd    przez, źródło, liczba
```

Specyficzne formy dla tych instrukcji to:

```
shl     reg, 1
shl     mem, 1
shl     reg,  imm
shl     mem, imm
shl     reg, cl
shl     mem, cl
```

sal jest synonimem dla shl i używa tych samych form.
shr używa tych samych form jak shl
sar używa tych samych form jak shl.



Rysunek 6.2: Operacja przesunięcia w lewo

```
shld    reg, reg, imm
shld    mem, reg, imm
shld    reg, reg, cl
shld    mem, reg, cl
```

Dla CPU 8088 i 8086 liczba bitów do przesunięcia to albo „1" albo wartość w cl. W 80286 i późniejszych procesorach możemy używać ośmio bitowej stałej bezpośredniej. Oczywiście wartość w cl lub stałą bezpośrednia powinny być mniejsze lub równe liczbie bitów w operandzie przeznaczenia. Byłoby marnotrawieniem czasu przesuwać w lewo dziewięć bitów(osiem stworzy ten sam wynik jak wkrótce zobaczymy. Algorytmicznie możemy myśleć o operacji przesunięcia z liczbą inną niż jeden jak następuje:

```
for temp := 1 do liczba do
        shift dest, 1
```

Jest drobna różnica w sposobie traktowania przez instrukcje przesunięcia flagi przepełnienia kiedy liczba nie jest jedynką, ale możemy to ignorować większość czasu.

Instrukcje shl ,sal ,shr i sar na ośmio- szesnasto- i trzydziesto dwu bitowych operandach. Instrukcje shld i shrd działają na 16 i 32 bitowych operandach.

6.6.2.1 SHL/SAL

Mnemoniki shl i sal są synonimami. Przedstawiają one te same instrukcje i używają identycznego binarnego kodowania. instrukcje te przenoszą każdy bit w operandzie przeznaczenia o jedną pozycję w lewo ilość razy wyszczególnioną w operandzie. Zera wypełniają opuszczone pozycje w najmniej znaczącym bicie; bardziej znaczący bit przesuwany jest do flagi przeniesienia (zobacz Rysunek 6.2)

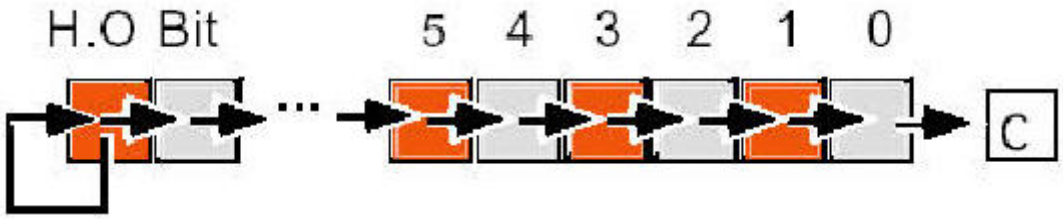
Instrukcja shl /sal ustawia bity kodu stanu jak następuje:

- Jeśli liczba przesunięcia wynosi zero, instrukcja shl nie wpływa na żadne flagi.
- Flaga przeniesienia zawiera ostatni bit przesunięty z bardziej znaczącego bitu operandu
- Flaga przepełnienia będzie zawierała jeden jeśli dwa bardziej znaczące bity były różne przed przesunięciem pojedynczego bitu. Flaga przepełnienia jest niezdefiniowana jeśli przesunięcie nie wynosi jeden.
- Flaga zera będzie wynosić jeden jeśli przesunięcie stworzy zero jako wynik.
- Flaga znaku będzie zawierała bardziej znaczący bit wyniku
- Flaga parzystości będzie zwierała jeden jeśli są parzyste liczby jedynek w najmniej znaczącym bajcie wyniku.
- Flaga A jest zawsze niezdefiniowana po instrukcji shl /sal.

Instrukcja przesunięcia w lewo jest zwłaszcza użyteczna dla danych upakowanych. Na przykład przypuśćmy, że mamy dwa nibble w al. i ah które chcemy połączyć. Możemy użyć następującego kodu do wykonaniu tego

```
shl     ah, 4           ;ta forma wymaga 80286 i późniejszego
or      al., ah         ; łączenie czterech bitów
```

Oczywiście al. musi zawierać wartość z zakresu 0..F dla tego kodu dla właściwej pracy (operacja przesunięcia w lewo automatycznie czyści mniej znaczące cztery bity ah przed instrukcją or).Jeśli bardziej znaczące cztery bity



Rysunek 6.3: Operacja Arytmetycznego Przesunięcia w Prawo

al nie są zerami ,przed tą operacją, możemy łatwo wyczyścić je instrukcją add:

```
shl     ah, 4           ;przenosi mniej znaczące bity na bardziej znaczące pozycje
and     al., 0Fh        ;czyści cztery bardziej znaczące bity
or      al., ah         ;łączy bity
```

Ponieważ przesuwanie wartości całkowitych w lewą stronę o jedną pozycję jest równoważne tej wartości przez dwa, możemy również użyć instrukcji przesuwania w lewo dla mnożenia przez potęgę dwóch:

```
shl     ax, 1           ;odpowiednik AX*2
shl     ax, 2           ;odpowiednik AX*4
shl     ax, 3           ;odpowiednik AX*8
shl     ax,4            ;odpowiednik AX*16
shl     ax, 5           ;odpowiednik  AX*32
shl     ax,6            ;odpowiednik AX*64
shl     ax,7            ;odpowiednik AX*128
shl     ax, 8           ;odpowiednik AX*256
```

Zauważmy, że shl ax,8 jest odpowiednikiem następujących instrukcji:

```
mov     ah, al.
mov     al., 0
```

Instrukcja shl /sal mnoży obie wartości ze znakiem i bez znaku przez dwa dla każdego przesunięcia. Ta instrukcja ustawia flagę przeniesienia jeśli wynik nie mieści się w operandzie przeznaczenia (tj. wystąpi bez znakowe przepełnienie).Podobnie, ta instrukcja ustawia flagę przepełnienia jeśli wynik ze znakiem nie mieści się w operandzie przeznaczenia .Wystąpi to kiedy przesuniemy zero do bardziej znaczącego bitu liczby ujemnej lub przesuniemy jeden do bardziej znaczącego bitu nie ujemnej liczby.

6.6.2.2 SAR

Instrukcja sar przesuwa wszystkie bity w operandzie przeznaczenia w prawo o jeden bit kopiując bardziej znaczący bit (zobacz Rysunek 6.3).

Instrukcja sar ustawia bity flag jak następuje:

- Jeśli liczba przesunięcia to zero, instrukcja sar nie wpływa na żadną flagę.
- Flaga przeniesienia zawiera ostatni bit przesunięty z mniej znaczącego bitu operandu.
- Flaga przepełnienia będzie zawierała zero jeśli przesunięcie to jeden. Przepełnienie może nigdy nie wystąpić z tą instrukcją. Jednakże, jeśli liczba ta to nie jeden, wartość flagi przepełnienia jest niezdefiniowana.
- Flaga zera będzie zawierała jeden jeśli przesunięcie tworzy wynik zero.
- Flaga znaku będzie zawierała najbardziej znaczący wyniku
- Flaga parzystości będzie zawierała jeden jeśli jest parzysta liczba jedynek w najmniej znaczącym bajcie wyniku.
- Flaga przepełnienia połówkowego jest zawsze niezdefiniowana po instrukcji sar.

Głównym celem wykonania instrukcja sar jest dzielenie ze znakiem przez potęgi dwójki. Każde przesunięcie w prawo dzieli wartość przez dwa. Wielokrotne przesunięcie w prawo dzieli poprzednią przesuniętą wartość przez dwa ,więc wielokrotne przesunięcie tworzy następujące rezultaty:

```
sar     ax, 1       ;Signed division by 2
sar     ax, 2       ;Signed division by 4
sar     ax, 3       ;Signed division by 8
sar     ax, 4       ;Signed division by 16
sar     ax, 5       ;Signed division by 32
sar     ax, 6       ;Signed division by 64
sar     ax, 7       ;Signed division by 128
sar     ax, 8       ;Signed division by 256
```

Jest bardzo ważna różnica pomiędzy instrukcjami sar i idiv. Instrukcja idiv zawsze zaokrągla do zera podczas gdy sar zaokrągla wynik do mniejszego wyniku. Dla wyników dodatnich, arytmetyczne przesunięcie w prawo o jedną pozycję tworzy taki sam wynik jak całkowite dzielenie przez dwa. Jednak, jeśli iloraz jest ujemny, instrukcja idiv zaokrągla do zera podczas gdy sar zaokrągla do ujemnej nieskończoności. Następujące przykłady demonstrują te różnice:

```
        mov     ax, -15
        cwd
        mov     bx, 2
        idiv                    ;daje –7
        mov     ax, -15
        sar     ax, 1           ;daje –8
```

Zapamiętajmy to jeśli używamy sar dla operacji dzielenia całkowitego.

Instrukcja sar ax, 8 faktycznie kopiuje ah do al. a potem powiela znak al. do ah. Jest tak ponieważ sar ax, 8 przesunie ah do al. ale pozostawi kopię najstarszego bitu ah na wszystkich pozycjach bitów ah .Istotnie, możemy użyć instrukcji sar w 80286 i późniejszych procesorach do powielenia znaku jednego rejestru do innego .Następująca sekwencja daje nam przykład takiego zastosowania:

```
;Odpowiednik  CBW:
        mov     ah, al.
        sar     ah, 7
;Odpowiednik CWD:
        mov     dx, ax
        sar     dx, 15
;Odpowiednik CDQ:
        mov     edx, eax
```

```
        sar     edx, 31
```

Oczywiście ,może wydawać się głupie użycie dwóch instrukcji tam gdzie może wystarczyć pojedyncza instrukcja; jednakże instrukcje cbw, cwd i cdq tylko powielają znak al. do ax, ax do dx:ax i eax do edx:eax. Instrukcja sar pozwala nam powielić znak jednego rejestru do innego rejestru o tym samym rozmiarze, z drugiego rejestru zawierającego powielony znak bitów:

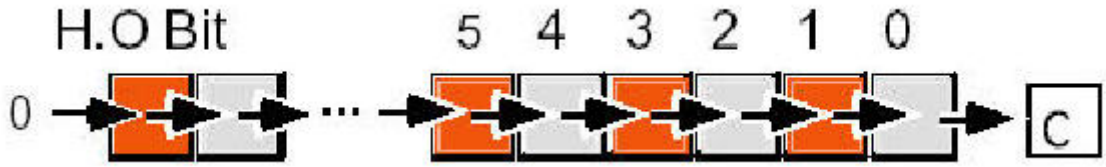
```
; Powielenie znaku bx do cx:bx
        mov     cx, bx
        sar     cx, 15
```

6.6.2.3 SHR

Instrukcja shr przesuwa wszystkie bity w operandzie przeznaczenia w prawo o jeden bit przesuwając zero do najbardziej znaczącego bitu (zobacz Rysunek 6.4)

Instrukcja shr ustawia bity flag jak następuje:

- Jeśli liczba do przesunięcia to zero, instrukcja shr nie wpływa na żadną flagę.
- Flaga przeniesienia zawiera ostatni bit przesunięty z najmniej znaczącego bitu operandu
- Jeśli liczba do przesunięcia to jeden, flaga przepełnienia będzie zawierała wartość najbardziej znaczącego bitu operandu przed przesunięciem. (tj. instrukcja ta ustawia flagę przepełnienia



Rysunek 6.4:Operacja przesunięcia w prawo

- Jeśli zmienia się znak).Jednakże ,jeśli liczba nie jest jedynką wartość flagi przepełnienia jest Niezdefiniowana
- Flaga zera będzie jedynką jeśli przesunięcie stworzy wynik zero
- Flaga znaku będzie zwierała bardziej znaczący bit wyniku, który jest zawsze zero
- Flaga parzystości będzie zawierała jedynkę jeśli jest parzysta liczba bitów jeden w najmniej znaczącym bajcie wyniku
- Flaga przeniesienia połówkowego jest zawsze niezdefiniowana po instrukcji shr

Instrukcja przesunięcia w prawo jest użyteczna zwłaszcza dla danych nie upakowanych. Na przykład przypuśćmy że chcemy uzyskać dwa nibble w rejestrze al., pozostawiając bardziej znaczący nibble w ah i najmniej znaczący nibble w al. Moglibyśmy używać następującego kodu do zrobienia tego:

```
        mov     ah, al.
        shr     ah, 4
        and     al., 0Fh
```

Ponieważ przesunięcie wartości całkowitej bez znaku w prawo o jedną pozycję jest odpowiednikiem dzielenia tej wartości przez dwa ,możemy również używać instrukcji przesunięcia w prawo dla dzielenia przez potęgę dwójki:

```
shr     ax, 1       ;Equivalent to AX/2
shr     ax, 2       ;Equivalent to AX/4
shr     ax, 3       ;Equivalent to AX/8
shr     ax, 4       ;Equivalent to AX/16
shr     ax, 5       ;Equivlaent to AX/32
shr     ax, 6       ;Equivalent to AX/64
shr     ax, 7       ;Equivalent to AX/128
shr     ax, 8       ;Equivalent to AX/256
etc.
```

Zauważmy, że shr ax, 8 jest odpowiednikiem następujących dwóch instrukcji;

```
        mov     al., ah
        mov     ah, 0
```

Pamiętajmy, że dzielenie przez dwa używa shr tylko działając dla operandów bez znakowych. jeśli ax zawiera –1 a mamy wykonać shr ax,1 wynik w ax będzie 32767 (7FFFh), nie –1 lub zero jak można by się spodziewać. Używamy instrukcji sar jeśli musimy podzielić wartość całkowitą ze znakiem przez potęgę dwójki.

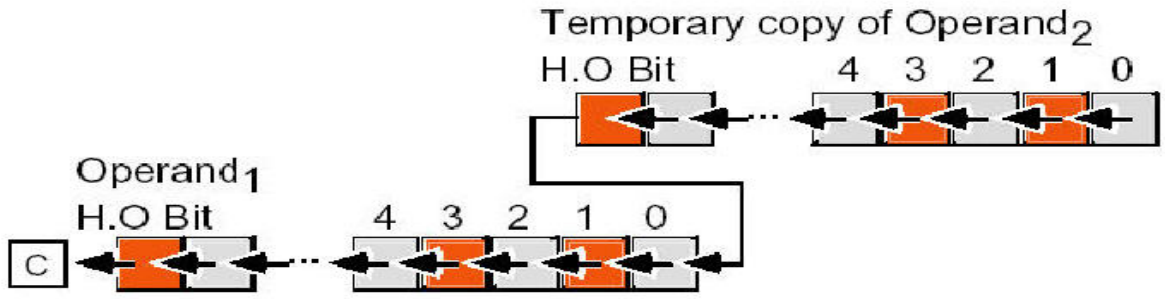
6.6.2.4 INSTRUKCJE SHLD I SHRD

Instrukcje shld i shrd dostarczają podwójnej precyzji operacji przesunięcia w lewo i prawo, odpowiednio. Te instrukcje są dostępne tylko na procesorach 80386 o późniejszych. Ich ogólna forma

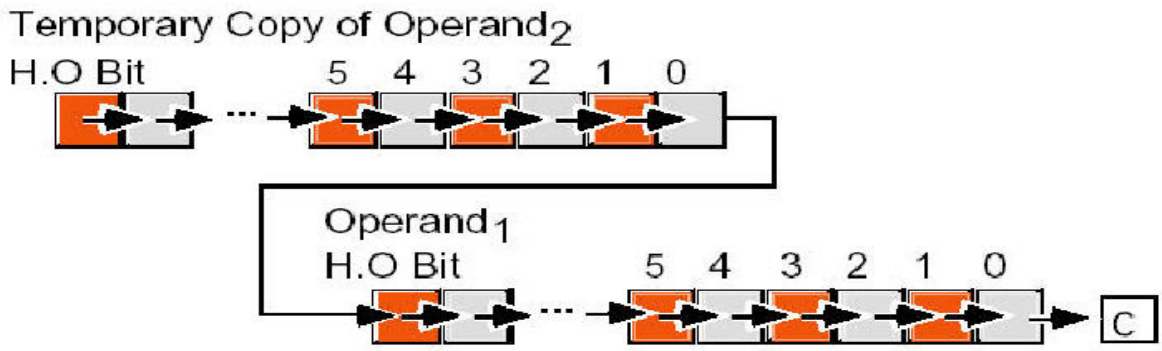
shld operand$_1$, operand$_2$, stała bezpośrednia
shld operand$_1$, operand$_2$, cl
shrd operand$_1$, operand$_2$, stała bezpośrednia
shrd operand$_1$,operand$_2$, cl

Operand$_2$ musi być szesnasto lub trzydziesto dwu bitowym rejestrem .Operand $_1$ może być rejestrem lub komórką pamięci. Oba operandy muszą być tego samego rozmiaru. Operand bezpośredni może być wartością z zakresu od zera do n-1,gdzie n jest liczbą bitów w dwóch operandach; wyszczególnia liczbę bitów do przesunięcia.

Instrukcja shld przesuwa bity w operandzie$_1$ w lewo. Najbardziej znaczący bit przesuwany jest do flagi przeniesienia, a najbardziej znaczący bit operandu$_2$ przesuwa się do najmniej znaczącego bitu operandu$_1$.Zauważmy,że ta instrukcja



Rysunek 6.5: Operacja przesunięcia w lewo z podwójną precyzją



Rysunek 6.6: Operacja przesunięcia w prawo z podwójną precyzją

nie modyfikuje wartości operandu$_2$,używa czasowej kopii operandu$_2$ podczas przesunięcia. Operand bezpośredni wyszczególnia liczbę bitów do przesunięcia. jeśli liczba to n, wtedy shld przesuwa bit n-1 do flagi przeniesienia. Przesuwa również n najbardziej znaczących bitów operandu$_2$ do n najmniej znaczących bitów operandu$_1$.Obrazowo,instrukcja shld wygląda tak jak na rysunku 6.5

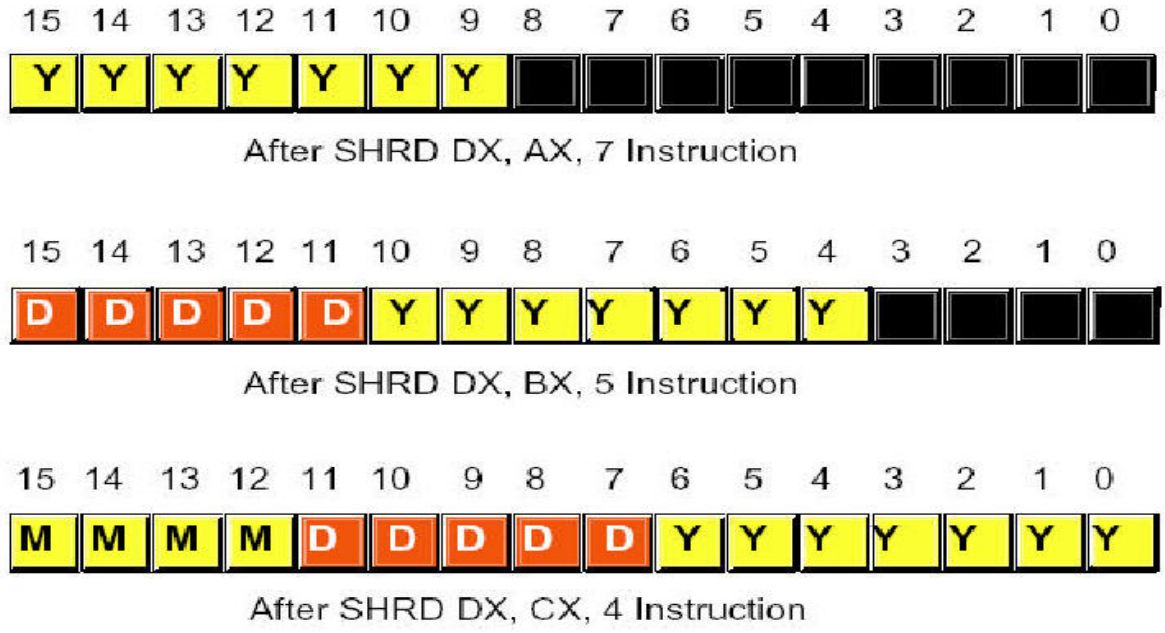
Instrukcja shld ustawia bity flag jak następuje:

- Jeśli liczba przesunięcia wynosi zero, instrukcja shld nie wpływa na żadną flagę
- Flaga przeniesienia zawiera ostatni bit przesunięty najbardziej znaczącego bitu operandu$_1$.
- Jeśli liczba przeniesienia to jeden ,flaga przepełnienia będzie zawierała jeden jeśli bit znaku operandu$_1$ zmieni się podczas przesunięcia. jeśli liczba nie jest jedynką, flaga przepełnienia jest niezdefiniowana.
- Flaga zera będzie miał jeden jeśli przesunięcie stworzy wynik zero.
- Flaga znaku będzie zawierała najbardziej znaczący bit wyniku

Instrukcja shld jest użyteczna dla danych upakowanych z wielu różnych źródeł. Na przykład przypuśćmy, że chcemy stworzyć słowo poprzez połączenie bardziej znaczących nibbli z czterech innych słów. Możemy to zrobić przy pomocy następującego kodu:

```
mov     ax, wartość4    ;pobieranie najbardziej znaczącego nibbla
shld    bx, ax, 4       ;kopiowanie bardziej znaczących bitów z AX do BX
mov     ax, wartość3    ;pobieranie nibbla #2
shld    bx, ax, 4       ;łączenie w bx
mov     ax, wartość2    ;pobieranie nibbla #1
shld    bx, ax, 4       ;łączenie w bx
mov     ax, wartość     ;pobieranie najmniej znaczącego nibbla
shld    bx,ax, 4        ;BX zawiera teraz wszystkie cztery nibble
```

Instrukcja shrd jest podobna do shld z wyjątkiem tego ,że przesuwa bity w prawo zamiast w lewo. Pokazuje to rysunek 6.6



Rysunek 6.7: Pakowanie danych instrukcją shrd

Instrukcja shrd ustawia bity flag jak następuje:

- Jeśli liczba przesunięcia wynosi zero, nie wpływa na żadną flagę.
- Flaga przeniesienia zawiera ostatni bit przesunięty z najmniej znaczącego bitu $operandu_1$.
- Jeśli liczba przesunięcia to jeden, flaga przepełnienia będzie zawierała jeden jeśli najbardziej znaczący bit $operandu_1$ się zmieni. Jeśli liczba nie jest jedynką, flaga przepełnienia jest niezdefiniowana.
- Flaga zera będzie jedynką jeśli przesunięcie stworzy wynik zero
- Flaga znaku będzie zawierała najbardziej znaczący bit wyniku.

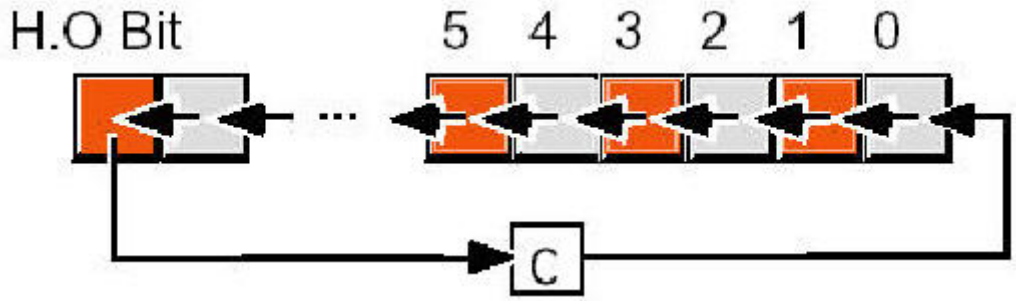
Szczerze mówiąc, te dwie instrukcje byłyby prawdopodobnie odrobinę bardziej użyteczne gdyby $Operand_2$ mógł być komórką pamięci. Intel stworzył te instrukcje pozwalające na szybkie przesunięcia (64 bity lub więcej) o zwiększonej dokładności. Po więcej informacji zajrzyj do „Operacje przesunięcia o rozszerzonej precyzji".

Instrukcja shrd jest tylko nieznacznie bardziej użyteczna niż shld dla pakowania danych. Na przykład przypuśćmy, że ax zawiera wartość z zakresu 0..99 przedstawiającą rok (1900..1999),bx zawiera wartość z zakresu 1..31 przedstawiającą dzień i cx zawierającą wartość z zakresu 1..12 przedstawiającą miesiąc (zobacz „Pola bitów i dane upakowane").Możemy łatwo użyć instrukcji shrd do pakowania tej danej do dx jak następuje:

```
shrd    dx,ax,7
shrd    dx,bx,5
shrd    dx,cx,4
```

6..6.3INSTRUKCJE OBROTU: RCL,RCR,ROL I ROR

Instrukcje obrotu przesuwają bity w koło, podobnie jak instrukcje przesunięcia, z wyjątkiem tego ,że przesunięte bity operandu przez instrukcje obrotu krążą po operandzie. Zawierają one rcl (obrót w lewo z uwzględnieniem flagi przeniesienia),rcr (obrót w prawo z uwzględnieniem flagi przeniesienia),rol (obrót w lewo) i ror (obrót w prawo).Wszystkie te instrukcje przyjmują następujące formy:



Rysunek 6.8: Operacja obrotu w lewo z uwzględnieniem flagi przeniesienia

```
rcl     przez, liczba
rol     przez, liczba
rcr     przez, liczba
ror     przez, liczba
```

Specjalne formy to:

```
rcl     reg, 1
rcl     mem, 1
rcl     reg, bezp
rcl     mem, bezp
rcl     reg, cl
rcl     mem, cl
```

rol ,rcr, ror używa tego samego formatu co rcl.

6.6.3.1 RCL

Rcl (obrót w lewo z uwzględnieniem flagi przeniesienia) ,jak sama nazwa wskazuje ,obraca bity w lewo z uwzględnieniem flagi przeniesienia i wraca do bitu zero po prawej stronie (zobacz Rysunek 6.8)

Zauważ, że jeśli obracamy z uwzględnieniem przeniesienia object n+1 razy, gdzie n jest liczbą bitów w obiekcie, zakończymy z oryginalną wartością .Zapamiętajmy jednak, że kilka flag może zawierać różne wartości po n+1 operacji rcl.

Instrukcja rcl ustawia bity flag jak następuje:

- Flaga przeniesienia zawiera ostatni bit przesunięty z najbardziej znaczącego bitu operandu
- Jeśli liczba przesunięcia to jeden, rcl ustawia flagę przepełnienia jeśli znak zmieni się jako wynik obrotu. Jeśli liczba nie jest jedynką, flaga przepełnienia jest niezdefiniowana.
- Instrukcja rcl nie modyfikuje flag zera, znaku, parzystości i przeniesienia połówkowego.

Ważna uwaga: W odróżnieniu od instrukcji przesunięcia, instrukcje obrotu nie wpływają na flagi znaku, zera ,parzystości lub przeniesienia połówkowego. Ten brak ortogonalności może sprawić nam dużo kłopotów jeśli zapomnimy o nim i spróbujemy przetestować te flagi po operacji rcl. Jeśli musimy przetestować jedną z tych flag po operacji rcl ,sprawdzimy najpierw flagi przeniesienia i przepełnienia (jeśli to konieczne) potem porównujemy wynik z zerem ustawiając inne flagi.

6.6.3.2 RCR

Instrukcja rcr (obrót w prawo z uwzględnieniem flagi przeniesienia) jest uzupełnieniem operacji instrukcji rcl Przesuwa bity w prawo z uwzględnieniem flagi przeniesienia i wraca z powrotem do najbardziej znaczącego bitu (zobacz rysunek 6.9)

Instrukcja ta ustawia flagi w porządku analogicznym do rcl:

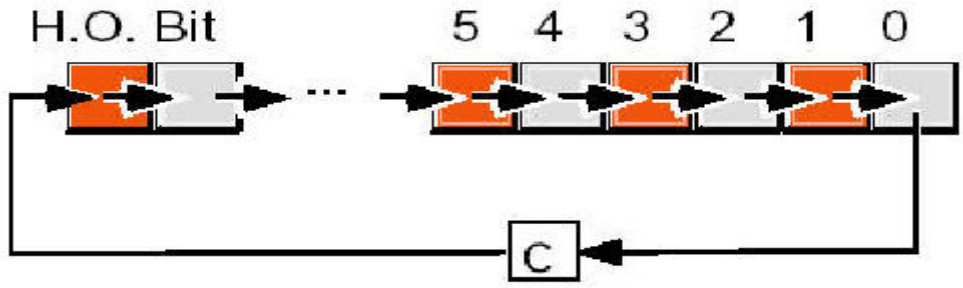
- Flaga przeniesienia zawiera ostatni bit przesunięty z najmniej znaczącego bitu operandu

- Jeśli liczba przesunięcia to jeden, rcr ustawia flagę przepełnienia jeśli znak zmieni się (w znaczeniu najbardziej znaczącego bitu a flaga przeniesienia nie była taka sama przed wykonaniem tej operacji)Jednak jeśli liczba nie jest jedynką, wartość flagi przepełnienia jest niezdefiniowana.
- Instrukcja rcr nie wpływa na flagi zera ,znaku ,parzystości lub przepełnienia połówkowego.

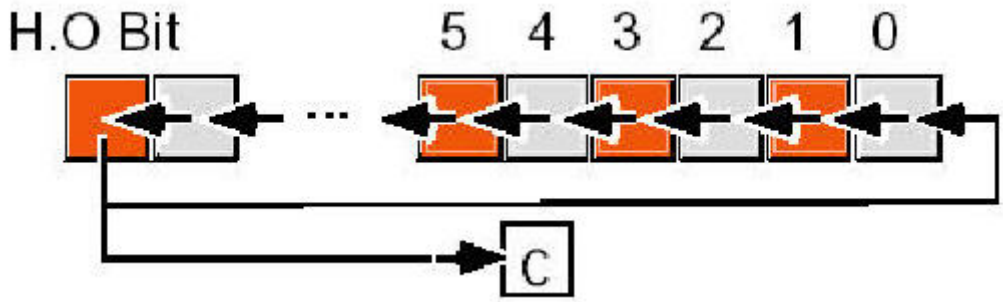
Instrukcji tej dotyczą te same uwagi jak powyższej instrukcji rcl

6.6.3.3 ROL

Instrukcja rol jest podobna do instrukcji rcl w tym, że obraca swój operand w lewo o określoną liczbę bitów. Główna różnica jest taka, że rol przesuwa najbardziej znaczący bit operandu zamiast przeniesienia do bitu zero.



Rysunek 6.9:Operacja obrotu w prawo z uwzględnieniem przeniesienia



Rysunek 6.10: Operacja obrotu w lewo

Rol również kopiuje wartość najbardziej znaczącego bitu do flagi przeniesienia (zobacz rysunek 6.10)

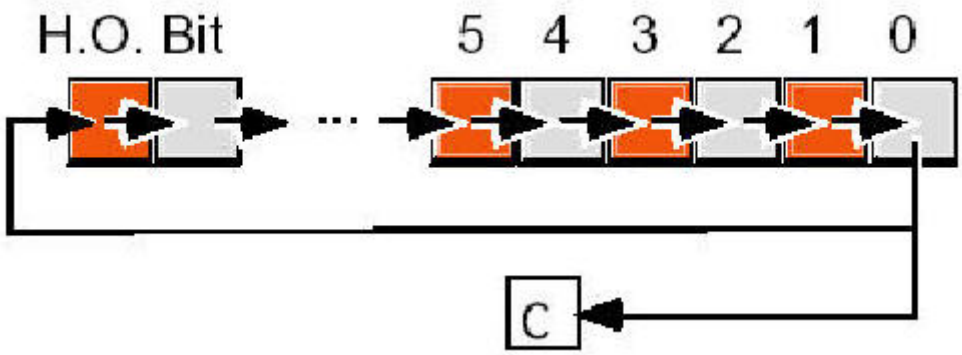
Instrukcja rol ustawia flagi identycznie do rcl.Za wyjątkiem wartości źródła przesuwanego do bitu zero, instrukcja ta zachowuje się jak instrukcja rcl. **Nie zapomnij ostrzeżenia o flagach!**

Podobnie jak shl, instrukcja rol jest często użyteczna dla pakowania i rozpakowania danych. Na przykład przypuśćmy ,że chcemy usunąć bity 10..14 w ax i pozostawić w bitach 0..4. Następująca sekwencja kodu osiągnie oba cele tak:

```
shr     ax, 10
and     ax, 1Fh
rol     ax, 6
and     ax, 1Fh
```

6.6.3.4 ROR

Instrukcja ror nawiązuje do instrukcji rcr w taki sam sposób jak instrukcja rol do instrukcji rcl. To znaczy, jest to prawie ta sama operacja z wyjątkiem bitu wejściowego źródła operandu. Zamiast przesunięcia poprzedniej flagi przeniesienia do bardziej znaczącego bitu operacji przeznaczenia, ror przesuwa bit zero do najbardziej znaczącego bitu (zobacz rysunek 6.11)



Rysunek 6.11:Operacja obrotu w prawo

Instrukcja ror ustawia flagi identycznie jak rcr. Z wyjątkiem bitu źródłowego przesuwanego do bardziej znaczącego bitu, instrukcja ta zachowuje się dokładnie jak instrukcja rcr. **Nie zapomnij uwagi o flagach!**

6.6.3 OPERACJE BITOWE

Zabawa z bitami jest jedną z tych łatwiejszych operacji do wykonania w języku asemblera niż w innych językach. I nic dziwnego .Większość języków wysokiego poziomu chroni nas przed przedstawianiem maszynowym odpowiednich typów danych. Instrukcje takie jak and ,or, xor ,not i obrotu i przesunięcia wykonują o ile to możliwe testowanie, ustawianie, zerowanie, odwracanie pól bitów wewnątrz łańcuchów bitów. Nawet C++ słynący ze swoich działań manipulowania bitami, nie dostarcza takich zdolności do manipulowania bitami jak asembler.

Procesory rodziny 80x86,zwłaszcza 80386 i późniejsze, idą dużo dalej. Poza standardowymi instrukcjami logicznymi, przesunięcia i obrotu, są instrukcje do testowania bitów wewnątrz operandu, do testowania i ustawiania, zerowania lub odwracania wyszczególnionych bitów w operandzie, i wyszukiwania dla zbioru bitów. Operacje te to:

```
test    przez, źródło
bt      źródło, indeks
btc     źródło, indeks
btr     źródło, indeks
bts     źródło, indeks
bsf     przez, źródło
bsr     przez, źródło
```

Formy określone:

```
test    reg, reg
test    reg, mem
test    mem, reg
test    reg, bezp
test    mem, bezp
test    eax,ax,al.,  bezp

bt      reg ,reg
bt      mem, reg
bt      reg, bezp
bt      mem, bezp

btc, btr i bts używają tego samego formatu co bt

bsf     reg, reg
bsf     reg, mem
```

Zauważmy, że bt, btc, btr, bts, bsf i bsr wymagają operandu 16- lub 32 bitowego. Operacje bitowe są użyteczne kiedy implementujemy zbiór typów danych używających mapy bitów.

6.6.4.1 TEST

Instrukcja test logicznie dodaje swoje dwa operandy i ustawia flagi, ale nie zapisuje rezultatu. Test i and dzieli ten sam związek co cmp i sub. Chcielibyśmy użyć tej instrukcji aby zobaczyć czy bit zawiera jeden. Rozważmy następująca instrukcję:

```
test  al., 1
```

Instrukcja ta doda logicznie al. z wartością jeden. Jeśli bit zero al zawiera jeden, wynik nie jest zerowy a 80x86 czyści flagę zera. Jeśli bit zero al zawiera zero wtedy wynik jest zerem a operacja test ustawia flagę zera. Możemy testować flagę zera po tej instrukcji aby zdecydować czy al zawierał zero lub jeden w bicie zero.

Instrukcja test może również sprawdzić czy jeden lub więcej bitów w rejestrze lub komórce pamięci nie jest zerem. Rozważmy następującą instrukcję:

```
test  dx, 105h
```

Instrukcja ta logicznie dodaje dx z wartością 105h.tworzy to nie zerowy wynik (i dlatego też czyści flagę zera) jeśli przynajmniej jeden z bitów zero, dwa lub osiem zawiera jeden. Wszystkie muszą mieć wartość zero do ustawiania flagi zera.

Instrukcja test ustawia flagi identycznie jak instrukcja and:

- Zeruje flagę przeniesienia
- Czyści flagę przepełnienia
- Ustawia flagę zera jeśli wynikiem jest zero, inaczej ją zeruje
- Kopiuje bardziej znaczący bit wyniku do flagi znaku
- Ustawia flagę parzystości wedle parzystości (liczby bitów jeden) w mniej znaczącym bajcie wyniku
- Zmienia flagę przeniesienia połówkowego.

6.6.4.2 INSTRUKCJE TESTOWANIA BITÓW: BT,BTS,BTR I BTC

W 80386 i późniejszych procesorach, możemy użyć instrukcji bt (bit test) do testowania pojedynczego bitu. Jej drugi operand wyszczególnia indeks bitu w pierwszym operandzie. Bt kopiuje adresowany bit do flagi przeniesienia. Na przykład, instrukcja

```
bt      ax, 12
```

kopiuje bit dwunasty z ax do flagi przeniesienia.

Instrukcje bt/bts/btr/btc wykorzystują operandy 16 lub 32 bitowe. To nie jest ograniczenie tej instrukcji. W końcu jeśli chcemy przetestować trzeci bit rejestru al., możemy łatwo przetestować trzeci bit rejestru ax. Z drugiej strony ,jeśli indeks jest większy niż rozmiar operand rejestru, wynik jest niezdefiniowany.

Jeśli pierwszy operand jest komórką pamięci, instrukcja bitowa testuje bit w danym offsecie pamięci, bez względu na wartość indeksu. Na przykład, jeśli bx zawiera 65 wtedy

```
bt      TestMe, bx
```

skopiuje bit z komórki TestMe+8 do flagi przeniesienia .Jeszcze raz, rozmiar operandu nie ma znaczenia. Praktycznie rzecz biorąc, operand pamięci jest bajtem i możemy przetestować każdy bit po bajcie z właściwym indeksem. Faktyczny bit testowany przez bt to pozycja bitu indeks mod 8 a offset pamięci adres efektywny + indeks/8.

Instrukcje bts, btr i btc również kopiują adresowany bit do flagi przepełnienia. Jednakże instrukcje te również ustawiają, resetują (zerują) lub dopełniają (odwracają) bit w pierwszym operandzie po skopiowaniu go do flagi przeniesienia. dostarczają operacji testuj i ustaw, testuj i zeruj, testuj i odwróć koniecznych dla kilku równoległych algorytmów. Instrukcje bt, bts, btr i btc nie wpływają na żadną inną flagę niż flaga przeniesienia.

6.6.4.3 WYSZUKIWANIE BITÓW: BSF I BSR

Instrukcje bsf i bsr szukają pierwszego lub ostatniego ustawionego bitu w 16 lub 32 bitowej wielkości. Ogólna forma tych instrukcji to

```
bsf     przez, źródło
bsr     przez, źródło
```

Bsf umiejscawia pierwszy ustawiony bit w operandzie źródłowym ,szukając od bitu zero do najbardziej znaczącego bitu. Bsr umiejscawia pierwszy ustawiony bit szukając od bardziej znaczącego bitu w dół do najmniej znaczącego bitu. Jeśli te instrukcje umiejscawiają jedynkę, zerują flagę zero i przechowują indeks bitu (0..31) w operandzie przeznaczenia. Jeśli operand źródłowy to zero, instrukcje te ustawiają flagę zera i przechowują nieokreśloną wartość w operandzie przeznaczenia.

Szukając dla pierwszego bitu zawierającego zero (zamiast jeden),robimy kopię operandu źródłowego i odwracamy go (używając not),potem wykonujemy bsf i bsr na tej odwróconej wartości. Flaga zera byłaby ustawiona po tej operacji jeśli nie byłoby bitów zero w oryginalnej wartości źródłowej, w przeciwnym razie operand przeznaczenia zawierałby pozycję pierwszego bitu zawierającego zero.

6.6.5 INSTRUKCJE WARUNKOWE

Instrukcje setcc ustawiają pojedynczy bajt operandu (rejestr lub komórka pamięci) na zero lub jeden w zależności od wartości w rejestrze flag Ogólny format dla instrukcji setcc to

setcc reg_8
setcc mem_8

Setcc przedstawia mnemonik pojawiający się w następującej tabeli. Instrukcje przechowują zero w odpowiednim operandzie jeśli warunek jest fałszywy, przechowują jeden w ośmio bitowym operandzie jeśli warunek jest prawdziwy.

Instruction	Description	Condition	Comments
SETC	Set if carry	Carry = 1	Same as SETB, SETNAE
SETNC	Set if no carry	Carry = 0	Same as SETNB, SETAE
SETZ	Set if zero	Zero = 1	Same as SETE
SETNZ	Set if not zero	Zero = 0	Same as SETNE
SETS	Set if sign	Sign = 1	
SETNS	Set if no sign	Sign = 0	
SETO	Set if overflow	Ovrflw=1	
SETNO	Set if no overflow	Ovrflw=0	
SETP	Set if parity	Parity = 1	Same as SETPE
SETPE	Set if parity even	Parity = 1	Same as SETP
SETNP	Set if no parity	Parity = 0	Same as SETPO
SETPO	Set if parity odd	Parity = 0	Same as SETNP

Tabela 28: Instrukcje Setcc, które testują flagi

Powyższe instrukcje setcc po prostu testują flagi bez żadnych innych powiązań do tej operacji. Możemy, na przykład, użyć setcc do sprawdzenia flagi przeniesienia po przesunięciu ,obrocie, testowaniu bitów lub operacji arytmetycznych. .Podobnie, możemy użyć instrukcji senz po instrukcji test do sprawdzenia wyniku.

Instrukcja cmp działa w synergii z instrukcją setcc. Bezpośrednio po operacji cmp flagi dostarczają informacji dotyczących względnych wartości tych operandów .Pozwalają nam zobaczyć czy jeden operand jet mniejszy niż, równy, większy niż lub ich kombinację.

Są dwie grupy instrukcji setcc które są bardzo użyteczne po operacji cmp. Pierwsza grupa zajmuje się wynikiem bez znakowego porównania, druga grupa zajmuje się wynikami porównania znakowego.

Instruction	Description	Condition	Comments
SETA	Set if above (>)	Carry=0, Zero=0	Same as SETNBE
SETNBE	Set if not below or equal (not <=)	Carry=0, Zero=0	Same as SETA
SETAE	Set if above or equal (>=)	Carry = 0	Same as SETNC, SETNB
SETNB	Set if not below (not <)	Carry = 0	Same as SETNC, SETAE
SETB	Set if below (<)	Carry = 1	Same as SETC, SETNAE
SETNAE	Set if not above or equal (not >=)	Carry = 1	Same as SETC, SETB
SETBE	Set if below or equal (<=)	Carry = 1 or Zero = 1	Same as SETNA
SETNA	Set if not above (not >)	Carry = 1 or Zero = 1	Same as SETBE
SETE	Set if equal (=)	Zero = 1	Same as SETZ
SETNE	Set if not equal (≠)	Zero = 0	Same as SETNZ

Tabela 29: Instrukcje Setcc dla porównania bez znakowego

Odpowiadająca tabela dla porównania znakowego to

Instruction	Description	Condition	Comments
SETG	Set if greater (>)	Sign = Ovrflw or Zero=0	Same as SETNLE
SETNLE	Set if not less than or equal (not <=)	Sign = Ovrflw or Zero=0	Same as SETG
SETGE	Set if greater than or equal (>=)	Sign = Ovrflw	Same as SETNL
SETNL	Set if not less than (not <)	Sign = Ovrflw	Same as SETGE
SETL	Set if less than (<)	Sign ≠ Ovrflw	Same as SETNGE
SETNGE	Set if not greater or equal (not >=)	Sign ≠ Ovrflw	Same as SETL
SETLE	Set if less than or equal (<=)	Sign ≠ Ovrflw or Zero = 1	Same as SETNG
SETNG	Set if not greater than (not >)	Sign ≠ Ovrflw or Zero = 1	Same as SETLE
SETE	Set if equal (=)	Zero = 1	Same as SETZ
SETNE	Set if not equal (≠)	Zero = 0	Same as SETNZ

Tabela 30: Instrukcje Setcc dla porównania znakowego

Instrukcje setcc są szczególnie wartościowe ponieważ mogą konwertować wynik porównania do wartości boolowskiej (prawda/fałsz lub 0/1).jest to szczególnie ważne kiedy tłumaczymy instrukcje z języków wysokiego poziomu jak Pascal lub C++ na asembler. Następujący przykład pokazuje jak użyć tych instrukcji w tym celu:

```
; Bool := A <= B
        mov     ax, A           ; zakładamy, że A i B są wartościami całkowitymi ze znakiem
        cmp     ax, B
        setle   Bool            ; Bool musi być zmienną bajtową.
```

Ponieważ instrukcje setcc zawsze tworzą zero lub jeden, możemy użyć wyników logicznego and i or do obliczenia złożonych wartości boolowskich:

```
; Bool := ((A  <=B) and (D = E) )or (F<>G)
        mov     ax, A
        cmp     ax, B
        setle   bl
        mov     ax, D
        cmp     ax, E
        sete    bh
        and     bl, bh
        mov     ax, F
        cmp     ax, G
        setne   bh
        or      bl, bh
        mov     Bool, bh
```

Po więcej szczegółów zobacz „Wyrażenia Logiczne (Boolowskie)”

Instrukcje setcc zawsze tworzą ośmio bitowy wynik ponieważ bajt jest najmniejszym operandem na którym działa 80x86.Jednakże możemy łatwo użyć instrukcji przesunięcia i obrotu do upakowania ośmio bitowej wartości w pojedynczym bajcie. Następujące instrukcje porównują osiem różnych wartości z zerem i kopiują „flagę zera” z każdego porównania do odpowiednich bitów w al:

```
        cmp     Val7, 0
        setne   al.             ;wprowadza  pierwszą wartość w bicie #0
        cmp     val6, 0         ;testuje wartość dla bitu #6
        setne   ah              ;kopiuje flagę zera do rejestru ah
        shr     ah, 1           ;kopiuje flagę zera do przeniesienia
        rcl     al., 1          ;przesuwa przeniesienie do bajtu wyniku
        cmp     Val5, 0         ;testuje wartość dla bitu #5
        setne   ah
        shr     ah, 1
        rcl     al., 1
```

```
        cmp     Val4, 0         ;test wartości dla bitu #4
        setne   ah
        shr     ah, 1
        rcl     al., 1
        cmp     Val3, 0         ;test wartości dla bitu #3
        setne   ah
        shr     ah, 1
        rcl     al.,1
        cmp     Val2, 0         ;test wartości dla bitu #2
        setne   ah
        shr     ah, 1
        rcl     al., 1
        cmp     Val1, 0         ;test wartości dla bitu #1
        setne   ah
        shr     ah, 1
        rcl     al., 1
        cmp     Val0, 0         ;test wartości dla bitu #0
        setne   ah
        shr     ah, 1
        rcl     al., 1
;Teraz AL. zawiera flagę zera z ośmiu porównań
```

6.7 INSTRUKCJE I/O

80x86 wspiera dwie instrukcje I/O: in i out. Przybierają one formę:

```
        in      eax/ax/al., port
        in      eax/ax/al., dx
        out     port, eax/ax/al.
        out     dx, eax/ax/al.
```

port jest wartością między 0 a 255.

80x86 dostarcza 65,536 różnych portów I/O (wymagających 16 bitowych adresów).Wartość powyższego portu jest wartością pojedynczego bajtu. Dlatego też, możemy tylko bezpośrednio adresować pierwsze 256 portów I/O w przestrzeni adresowej I/O. Aby zaadresować wszystkie 65,536 różnych portów I/O, musimy załadować adres żądanego portu do rejestru dx i uzyskać dostęp do portu pośrednio. Instrukcja in odczytuje dane z wyszczególnionego portu i kopiuje je do rejestru akumulatora. instrukcja out zapisuje wartość z rejestru akumulatora do wyszczególnionego portu.

Proszę zauważyć ,że nie ma nic magicznego w instrukcjach in i out 80x86.Są one po prostu inną formą instrukcji mov która uzyskuje dostęp do innej przestrzeni pamięci (przestrzeń adresowa I/O) zamiast normalnej 1 Megabajtowej przestrzeni adresowej pamięci.

Instrukcje in i out nie wpływają na żadną flagę 80x86

Przykłady instrukcji I/O 80x86:

```
        in      al., 60h        ;odczytuje port klawiatury
        mov     dx, 378h        ;wskazuje na LPT1: port danych
        in      al., dx         ;odczytuje dane z portu drukarki
        inc     ax              ;zmniejsza ASCII kod o jeden
        out     dx, al.         ;zapisuje dane w AL. do portu drukarki
```

6.8 INSTRUKCJE ŁAŃCUCHOWE

890x86 dostarcza dwanaście instrukcji łańcuchowych:

- movs (przesuń łańcuch)
- lods (ładuj element łańcucha do rejestru akumulatora)
- stos (przechowaj akumulator w elemencie łańcucha)
- scas (szukanie łańcucha i sprawdzanie podobieństwa wartości w rejestrze akumulatora
- cmps (porównaj dwa łańcuchy)
- ins (wprowadzenie łańcucha z portu I/O)
- outs (wprowadzenie łańcucha do poru I/O)
- rep (powtarzaj operacje łańcuchowe)
- repz (powtarzaj dopóki zero)
- repe (powtarzaj dopóki równe)
- repnz (powtarzaj dopóki nie zero)

- repne (powtarzaj dopóki nie różne)

Możemy użyć instrukcji movs, stos, scas ,cmps ,ibns i outs do manipulowania pojedynczym elementem (bajt, słowo lub podwójne słowo) w łańcuchu, lub przetwarzać cały łańcuch. generalnie, moglibyśmy używać instrukcji lods do manipulowania pojedyncza pozycją.

Instrukcje te mogą działać na łańcuchach bajtów, słów lub podwójnych słów .Dla wyspecyfikowania rozmiaru obiektu, po prostu dodajemy b, w lub d na koniec mnemonika instrukcji, np. lodsb, movsw, cmpsd itp. Oczywiście, formy podwójnego słowa są tylko dostępne na procesorach 80386 i późniejszych.

Instrukcje movs i cmps zakładają, że ds:si zawiera adres segmentowy łańcucha źródłowego a es:di zawiera adres segmentowy łańcucha przeznaczenia. Instrukcja lods zakłada, że ds:si wskazuje łańcuch źródłowy, akumulator (al./ax/eax0 jest lokacja przeznaczenia. Instrukcje scas i stos zakładają, że es:di wskazuje łańcuch przeznaczenia a akumulator zawiera wartość źródłową.

Instrukcja mobs przesuwa jeden element łańcucha (bajt, słowo lub podwójne słowo) z komórki pamięci ds.:si do es:di.Po przesunięciu danych, instrukcja zwiększa lub zmniejsza si i di o jeden, dwa lub cztery jeśli przetwarzamy odpowiednio bajt, słowo lub podwójne słowo. CPU zwiększa te rejestry jeśli flaga kierunku jest wyzerowana, zmniejsza jeśli flag kierunku jest ustawiona.

Instrukcja movs może przesuwać bloki danych w pamięci. Możemy użyć jej do przesuwania łańcuchów, tablic i innych wielobajtowych struktur danych.

```
movs {b, w, d}:  es:[di]  := ds.:[si]
                 if flaga_kierunku= 0 then
                          si := si + rozmiar;
                          di := di + rozmiar;
                 else
                          si := si – rozmiar;
                          di := di – rozmiar;
                 endif;
```

Notka: rozmiar to jeden dla bajtu, dwa dla słowa i cztery dla podwójnego słowa.

Instrukcja cmps porównuje bajt, słowo lub podwójne słowo z lokacji ds.:si i es:di i ustawia odpowiednio flagi procesora. Po porównaniu, cmps zwiększa lub zmniejsza si i di o jeden, dwa lub cztery w zależności od rozmiaru instrukcji i stanu flagi kierunku w rejestrze flag.

```
cmps{b,w,d}:     cmp ds.:[si], es:[di]
                 if flaga_kierunku = 0 then
                          si :=si+ rozmiar;
                          di := di + rozmiar;
                 else
                          si := si – rozmiar;
                          di := di – rozmiar;
                 endif;
```

Instrukcja lods przesuwa bajt, słowo lub podwójne słowo spod ds.:si do rejestru al.,ax lub eax. Potem zwiększa lub zmniejsza rejestr si o jeden, dwa lub cztery w zależności od rozmiaru instrukcji i wartości flagi kierunku. Instrukcja lods jest użyteczna dla pobierania sekwencji bajtów, słów lub podwójnych słów z tablicy, wykonując jakieś operacje na tych wartościach a potem przetwarza następny element z łańcucha.

```
lods {b,w,d}:    eax,ax,al. := ds.:[si]
                 if flaga _kierunku = 0 then
                          si := si+rozmiar;
                 else
                          si := si – rozmiar;
                 endif;
```

Instrukcja stos przechowuje al ,ax lub eax pod adresem wyszczególnionym przez es:di. Di jest zwiększane lub zmniejszane według rozmiaru instrukcji i wartości flagi kierunku. Instrukcja stos ma kilka zastosowań. W parze z instrukcją lods może ładować (przez lods),manipulować i przechowywać elementy łańcucha. Sama, instrukcja stos może szybko przechować pojedynczą wartość w całej wielobajtowej strukturze danych.

```
stos{b,w,d}:     es:[di] ;+ eax/ax/al
                 if flaga_kierunku = 0 then
                          di := di +rozmiar;
                 else
                          di := di-rozmiar;
```

```
        endif;
```

Instrukcja scas porównuje al., ax lub eax z wartością spod lokacji es:di i potem modyfikuje odpowiednio di. Instrukcja ta ustawia flagi w rejestrze stanu procesora podobnie jak instrukcje cmp i cmps. instrukcja scas jest dobra dla szukania szczególnych wartości w całej wielobajtowej strukturze danych.

```
scas{b,w,d}:    cmp eax/ax/al., es:[di]
                if flaga_kierunku = 0 then
                        di  := di+ rozmiar;
                else
                        di := di – rozmiar;
                endif;
```

Instrukcja ins wprowadza bajt, słowo lub podwójne słowo z portu I/O wyszczególnionego w rejestrze dx. Przechowuje potem wartość wprowadzoną w komórce pamięci es:di i zwiększa lub zmniejsza di odpowiednio. Instrukcja ta jest dostępna tylko na procesorach 80286 i późniejszych.

```
ins {b,w,d}:    es:[di]  := port(dx)
                if flaga_kierunku = 0 wtedy
                        di := di+ rozmiar;
                else
                        di := di + rozmiar
                endif;
```

Instrukcja outs pobiera bajt, słowo lub podwójne słowo spod adresu ds:si, zwiększając lub zmniejszając odpowiednio si, a potem wprowadza wartość do portu wyszczególnionego w rejestrze dx.

```
outs{b,w,d]:    port(dx) := ds.:[si]
                if flaga_kierunku = 0 then
                        si := si + rozmiar;
                else
                        si :=  si – rozmiar;
                endif;
```

Jak wyjaśniliśmy tu ,instrukcje łańcuchowe są użyteczne, ale może być jeszcze lepiej .Kiedy łączymy je z przedrostkiem rep ,repz, repe, repnz i repne, pojedyncza instrukcja może działać na całym łańcuchu.

6.9 INSTRUKCJE STEROWANIA PRZEPŁYWEM DANYCH W PROGRAMIE

Instrukcje omawiane do tej pory wykonywały się sekwencyjnie; to znaczy, CPU wykonuje każdą instrukcję w tej kolejności w jakiej pojawiają się w naszym programie. Napisanie rzeczywistych programów wymaga kilku struktur sterujących, nie sekwencyjnych. Przykłady obejmują instrukcję if, pętle i wywołanie podprogramu. Ponieważ kompilatory redukują wszystkie inne języki do języka asemblera, nie powinno być dla nas niespodzianką ,że asembler wspiera niezbędne instrukcje do implementacji tych struktur sterujących. Instrukcje sterujące programem 80x86 należą do trzech grup: bezwarunkowego przekazania sterowania ,warunkowego przekazania sterowania i wywołania podprogramu i instrukcji powrotu Następna sekcja omawia te instrukcje.

6.9.1 SKOKI BEZWARUNKOWE

Instrukcja jmp (skok) przenosi sterowanie bezwarunkowo do innego punktu w programie. Jest sześć form tej instrukcji: międzysegmentowy /skok bezpośredni, dwa wewnątrz segmentowe/ skoki bezpośrednie, międzysegmentowy /skok pośredni, dwa wewnątrzsegmentowe /skoki pośrednie. Skoki wewnątrzsegmentowe są zawsze pomiędzy instrukcjami w tym samym segmencie kodu. Skoki międzysegmentowe mogą przenosić sterowanie do instrukcji do różnych segmentów kodu.

Instrukcje te używają tej samej składni:

```
        jmp     cel
```

Asembler rozróżnia je po ich operandach:

```
jmp     disp8     ;bezpośrednio wewnątrz segmentu, 8 bitowe przesunięcie
jmp     disp16    ;bezpośrednio wewnątrz segmentu,16 bitowe przesunięcie
jmp     adrs32    ;bezpośrednio między segmentami, 32 bitowy adres segmentowy
jmp     mem16     ;pośrednio wewnątrz segmentu, 16 bitowy operand pamięci
jmp     reg16     ;pośrednio wewnątrz segmentu
jmp     mem32     ;pośrednio między segmentami, 32 bitowy operand pamięci.
```

Międzysegmentowy jest synonimem dla daleko, wewnątrzsegmentowy jest synonimem dla blisko.

Dwa bezpośrednie skoki wewnątrzsegmentowe różnią się tylko ich długościami. Pierwsza forma skalda się z opcodu i pojedyncze bajtowe przesunięcie. CPU powiela znak tego przesunięcia do 16 bitów i dodaje go do rejestru ip. Instrukcja ta może rozgałęziać się do lokacji –128..+127 od początku następującej instrukcji następującej po niej.

Druga forma wewnątrzsegmentowego skoku jest długa na trzy bajty z dwu bajtowym przesunięciem. Instrukcja ta pozwala na faktyczny zakres –32,768..+32,767 bajtów i może sterować przepływem gdzieś w bieżącym segmencie kodu .CPU po prostu dodaje dwa bajty przesunięcia do rejestru ip.

Te pierwsze dwa skoki używają względnego schematu adresowania. Offset kodowany jako część bajtu opcodu nie jest adresem docelowym w bieżącym segmencie kodu, ale odległością od adresu docelowego. Na szczęście MASM obliczy tą odległość za nas automatycznie, więc nie musimy obliczać wartości tego przesunięcia osobiście. Pod wieloma względami te instrukcje są niczym więcej niż instrukcjami add ip, disp.

Bezpośredni skok międzysegmentowy jest długości pięciu bajtów ,ostatnie cztery bajty zawierają adres segmentowy(offset w drugim i trzecim bajcie, segment w czwartym i piątym bajcie)Instrukcja ta kopiuje offset do rejestru ip a segment do rejestru cs. Wykonywanie następnej instrukcji zaczyna się od nowego adresu w cs:ip. W odróżnieniu od poprzednich dwóch skoków, adres opcodu jest absolutnym adresem pamięci instrukcji docelowej; ta wersja nie używa względnego adresowania .Instrukcja ta ładuje cs:ip 32 bitową wartością bezpośrednią.

Dla tych trzech skoków bezpośrednich opisanych powyżej, zwykle wyszczególniamy adres docelowy używając etykiety instrukcji. Etykieta instrukcji jest zazwyczaj identyfikatorem następującym przed dwukropkiem, zazwyczaj w tej samej linii jak wykonywana instrukcja maszynowa. Asembler określa offset instrukcji po etykiecie i automatycznie oblicza odległość od instrukcji skoku do etykiety instrukcji. Dlatego też, nie musimy martwić się o ręczne obliczanie przesunięcia. Na przykład, następująca krótka pętla stale odczytuje dane portu równoległego drukarki i odwraca najmniej znaczący bit. Tworzy to falę prostokątną sygnału elektrycznego na jednej z linii wyjściowych portu drukarki:

```
                mov     dx, 378h        ;adres portu równoległego drukarki
LoopForever:    in      al., dx         ;odczytanie znaku z portu wyjściowego
                xor     al., 1          ;odwrócenie najmniej znaczącego bitu
                out     dx, al.         ;dana wyjściowa wraca do portu
                jmp     LoopForever     ;powtórka
```

Czwarta forma instrukcji skoku bezwarunkowego jest to instrukcja skoku pośredniego wewnątrzsegmentowego. Wymaga ona 16 bitowego operandu pamięci. Forma ta steruje przepływem do dresu wewnątrz offsetu danego przez dwa bajty operandu pamięci. Na przykład,

```
WordVar         word    AdresDocelowy
                -
                -
                -
                jmp     WordVar
```

steruje przepływem danych do adresu wyszczególnionego przez wartość w 16 bitowej komórce pamięci WordVar. Nie jest to skok do instrukcji pod adresem WordVar, skacze do instrukcji pod adresem mieszczącym się w zmiennej WordVar. Zauważmy, że ta forma instrukcji jmp jest mniej więcej odpowiednikiem:

```
                mov ip, WordVar
```

Chociaż powyższy przykład używa zmiennej z pojedynczym słowem zawierającym adres pośredni, możemy użyć każdego ważnego trybu adresowania pamięci, a nie tylko trybu adresowania „tylko przemieszczenie”. Możemy użyć pośredniego trybu adresowania pamięci jak:

```
                jmp     DispOnly                ;zmienna Słowo
                jmp     Disp[bx]                ;disp jest tablicą słów
                jmp     Disp[bx][si]
                jmp     [bx]
                itd.
```

Rozważymy indeksowy tryb adresowania za chwilę (disp[bx])Ten tryb adresowania pobiera słowo z lokacji disp+bx i kopiuje tą wartość do rejestru ip; pozwala to nam stworzyć tablicę wskaźników i skoczyć do wyszczególnionego wskaźnika używając indeksu tablicy.

Rozważmy następujący przykład:

```
AdrsArray       word    stmt1,stmt2,stmt3,stmt4
                -
                -
                -
                mov     bx, I           ;I jest z zakresu 0..3
                add     bx, bx          ;indeks do tablicy słów
                jmp     AdrsArray[bx]   ;skok do stmt1,stmt2 itd., w zależności od wartości I
```

Ważną rzeczą do zapamiętania jest to ,że bliski skok pośredni pobiera słowo z pamięci i kopiuje go do rejestru ip; nie skacze do wyszczególnionej komórki pamięci, skacze pośrednio przez 16 bitowy wskaźnik spod wyszczególnionej komórki pamięci.

Piąta instrukcja jmp steruje przepływem offsetu danego w 16 bitowym rejestrze ogólnego przeznaczenia. Zauważmy ,że możemy użyć każdego rejestru ogólnego przeznaczenia, nie tylko bx ,si, di lub bp. Instrukcja w formie

```
        jmp     ax
```

jest mniej więcej odpowiednikiem

```
        mov     ip, ax
```

Zauważmy, że poprzednie dwie formy (rejestr lub pamięć pośrednia) są rzeczywiście tymi samymi instrukcjami. Pole mod and r /m. bajtu mod-reg-r/m. wyszczególniają adres pośredni rejestru lub pamięci. Zobacz Appendix D po więcej szczegółów.

Szósta forma instrukcji jmp, pośredni skok międzysegmentowy, ma operand pamięci który zawiera wskaźnik na podwójne słowo. CPU kopiuje podwójne słowo spod tego adresu do pary rejestrów cs:ip. Na przykład,

```
FarPointer      dword   AdresDocelowy
                -
                -
                -
                jmp     FarPointer
```

steruje przepływem danych do adresu segmentowego wyszczególnionego przez cztery bajty adresu FarPointer. Instrukcja ta jest semantycznie identyczna do (mitycznej) instrukcji

```
        lcs     ip, FarPointer          ;ładuje cs, ip z FarPointer
```

Ponieważ dla bliskiego skoku pośredniego omówionego wcześniej, ten daleki skok pośredni pozwala nam wyszczególnić każdy ważny tryb adresowania. Nie jesteśmy ograniczenia to trybu adresowania „tylko przemieszczenie" używanych przez powyższe przykłady.

MASM używa bliskiego pośredniego i dalekiego pośredniego trybu adresowania w zależności od typu komórki pamięci którą wyszczególnimy .Jeśli zmienna którą wyszczególniamy jest zmienna słowo ,MASM automatycznie wygeneruje bliski skok pośredni; jeśli zmienna jest podwójnym słowem, MASM wyemituje opcod dla dalekiego skoku pośredniego. Pewne formy adresowania pamięci niestety nie wyszczególniają samoistnie rozmiaru. Na przykład [bx] jest zdecydowanie operandem pamięci, ale czy punkt bx jest zmienną słowa czy podwójnego słowa? Może wskazywać jeden i drugi. Dlatego też MASM odrzuci instrukcję w tej formie:

```
        jmp     [bx]
```

MASM nie może powiedzieć czy to powinien być skok bliski pośredni czy daleki pośredni. Aby rozwiązać tę dwuznaczność, będziemy musieli użyć operatora sprawdzania zgodności typów. Rozdział Osiem w pełni opisuje operator sprawdzania zgodności typów, ale teraz ,użyjemy jednej z następujących dwóch instrukcji dla bliskiego lub dalekiego skoku, odpowiednio:

```
        jmp     word ptr [bx]
        jmp     dword ptr [bx]
```

Tryb adresowania pośredniego przez rejestr nie jest jedynym, który mógłby być typem dwuznacznym. Możemy również podejść do tego problemu z trybem adresowania indeksowym i bazowym indeksowym:

```
        jmp     word ptr 5[bx]
        jmp     dword ptr 9 [bx][si]
```

Więcej o operatorach sprawdzania zgodności typów znajdziesz w Rozdziale Ósmym.

Teoretycznie, możemy użyć instrukcji skoku pośredniego i instrukcji setcc do warunkowego sterowania przesyłaniem danych kilku danych lokacji. Na przykład następujący kod steruje przesyłaniem danych do iftrue jeśli zmienna słowa X jest równa zmiennej słowa Y.W przeciwnym razie steruje przesyłaniem danych do iffalse.

```
JmpTbl          word    iffalse, iftrue
                -
                -
                -
                mov     ax, X
                cmp     ax, Y
                sete    bl
                movzx   ebx, bl
                jmp     JmpTbl [ebx*2]
```

Jak zobaczymy wkrótce jest dużo lepszy sposób zrobienia tego używając instrukcji skoku warunkowego.

6.9.2 INSTRUKCJE CALL I RET

Instrukcje call i ret obsługują wywołania podprogramów i powroty. Jest pięć różnych instrukcji call i sześć różnych form instrukcji powrotu:

```
call    disp16          ;bezpośrednio wewnątrz segmentu
call    adrs32          ;bezpośrednio między segmentami, 32 bitowy adres segmentowy
call    mem16           ;pośrednio wewnątrz segmentu,16 bitowy wskaźnik pamięci
call    reg16           ;pośrednio wewnątrz segmentu,16 bitowy wskaźnik rejestru
call    mem32           ;pośrednio między segmentamil,32 bitowy wskaźnik pamięci

ret                     ;bliski lub daleki powrót
retn                    ;bliski powrót
retf                    ;daleki powrót
ret     disp            ;bliski lub daleki powrót i zdjęcie ze stosu
retn    disp            ;bliski powrót i zdjęcie ze stosu
retf    disp            ;daleki powrót i zdjęcie ze stosu
```

Instrukcje call przybierają takie same formy jak instrukcje jmp z wyjątkiem tego, że nie są krótsze (dwa bajty) wywołania wewnątrzsegmentowego.

Daleka instrukcja call robi co następuje:

- Odkłada rejestr cs na stos
- Odkłada 16 bitowy offset następnej instrukcji po wywołaniu na stos
- Kopiuje 32 bitowy adres efektywny do rejestrów cs:ip. Ponieważ instrukcja call pozwala na to, że te same tryby adresowania jak jmp, call mogą uzyskiwać adres docelowy używając względnego, pamięci lub rejestru, trybu adresowania.
- Kontynuuje się wykonywanie pierwszej instrukcji podprogramu. Ta pierwszą instrukcją jest opcod adresu docelowego obliczonego w poprzednim kroku.

Bliska instrukcja call robi co następuje:

- Odkłada 16 bitowy offset następnej instrukcji po wywołaniu na stos
- Kopiuje 16 bitowy adres efektywny do rejestru ip. Ponieważ instrukcja call pozwala na to, że te same tryby adresowania jak jmp, call mogą uzyskiwać adres docelowy używając względnego, pamięci lub rejestru, trybu adresowania.
- Kontynuowane jest wykonywanie pierwszej instrukcji podprogramu. Tą pierwszą instrukcją jest opcod adresu docelowego obliczonego w poprzednim kroku.

Instrukcja call $disp_{16}$ używa adresowania względnego. Możemy obliczyć adres efektywny poprzez dodanie tego 16 bitowego przesunięcia z adresem powrotnym (podobnie jak względna instrukcja jmp, przesunięcie jest odległością od instrukcji następującej po call do adresu docelowego).

Instrukcja call $disp_{32}$ używa bezpośredniego trybu adresowania.32 bitowy adres segmentowy bezpośrednio następuje po opcodzie call. Forma ta instrukcji call kopiuje ta wartość bezpośrednio do pary rejestrów cs:ip. Pod wieloma względami, jest to odpowiednik bezpośredniego trybu adresowania ponieważ wartość tej instrukcji jest kopiowana do pary rejestrów bezpośrednio następujących po tej instrukcji.

Call mem_{16} używa pośredniego trybu adresowania pamięci. Podobnie jak instrukcja jmp, ta forma instrukcji call pobiera słowo spod wyszczególnionej komórki pamięci i używa wartości tego słowa jako adresu docelowego. Pamiętamy, że możemy użyć każdego trybu adresowania pamięci z tą instrukcją. Tryb adresowani „tylko przesunięcie" jest najbardziej powszechna formą, ale inne są równie ważne:

```
call    CallTbl [bx]        ;indeks do  tablicy wskaźników
call    word ptr [bx]       ;BX wskazuje słowo do użycia
call    WordTbl [bx][si]    ;itd.
```

Zauważmy, że wybieranie trybu adresowania tylko wpływa na obliczenie adresu efektywnego dla podprogramu docelowego. Te instrukcje call odkładają offset następnej instrukcji następującej po call na stos. Ponieważ są to bliskie wywołania (uzyskują swój adres docelowy z 16 bitowej komórki pamięci) odkładają 16 bitowy adres powrotny na stos.

Call reg_{16} pracuje jak powyższe pośrednie wywołanie pamięci ,z wyjątkiem tego, że używamy 16 bitowej wartości w rejestrze dla adresu docelowego. Instrukcja ta jest tą samą instrukcją jak instrukcja call mem_{16}.Obie formy wyszczególniają swój adres efektywny używając bajtu mod-reg-r/m. Dla postaci call reg_{16},bit mod zawiera 11b więc pole r/m. specyfikuje rejestrowy zamiast pamięciowy tryb adresowania. Oczywiście ,instrukcja ta również odkłada 16 bitowy offset następnej instrukcji na stos jako adres powrotny.

Instrukcja call mem_{32} jest pośrednim dalekim wywołaniem. Adres pamięci wyszczególniony przez tą instrukcję musi być wartością podwójnego słowa. Ta forma instrukcji call pobiera 32 bitowy adres segmentowy, oblicza adres efektywny i kopiuje tą wartość podwójnego słowa do pary rejestrów cs:ip. Instrukcja ta również kopiuje 32bitowy adres segmentowy następnej instrukcji na stos (odkłada najpierw wartość segmentu a

potem offset).Podobnie jak instrukcją call mem16,możemy użyć każdego ważnego trybu adresowania pamięci z tą instrukcją:

```
call    DWordVar
call    DwordTbl [bx]
call    dword ptr [bx]
itd.
```

Jest względnie łatwo zsyntetyzować instrukcję call używając dwóch lub trzech innych instrukcji 80x86.Możemy stworzyć odpowiednik bliskiego wywołania używając instrukcji push i jmp:

```
push    <offset instrukcji po jmp>
jmp     podprogram
```

Dalekie wywołanie będzie podobne, będziemy musieli dodać instrukcję push cs przed tymi dwoma instrukcjami aby odłożyć daleki adres powrotu na stos.

Instrukcja ret (powrót) jest instrukcją powrotu z podprogramu. Robi to przez zdjęcie adresu powrotu ze stosu i steruje przesyłaniem danych do instrukcji spod tego adresu powrotu. Wewnątrzsegmentowy (bliski) powrót zdejmuje 16 bitowy adres powrotu ze stosu do rejestru ip. Międzysegmentowy (daleki) powrót zdejmuje 16 bitowy offset do rejestru ip a potem 16 bitową wartość segmentu do rejestru cs. Instrukcje te są faktycznie równe następującym:

```
retn    pop ip
retf    popd cs:ip
```

Musimy wyraźnie dopasować bliskie wywołanie podprogramu z bliskim powrotem a dalekie wywołanie podprogramu z odpowiadającym mu dalekim powrotem. Jeśli wymieszamy bliskie wywołanie z dalekim powrotem lub vice versa, zostawimy stos w niespójnym stanie i prawdopodobnie nie powrócimy do właściwej instrukcji po wywołaniu. Oczywiście, inną ważną kwestia kiedy używamy instrukcji call i ret jest to, że musimy upewnić się, że nasz podprogram nie odkłada czegoś na stos a potem spotka go niepowodzenie przed próbą powrotu z wywołania. Problemy ze stosem są główną przyczyną błędów w podprogramach języka asemblera. Rozważmy następujący kod:

```
Podprogram:  push    ax
             push    bx
             -
             -
             -
             pop     bx
             ret
             -
             -
             -
             call Podprogram
```

Instrukcja call odkłada na stos adres powrotu na stos a potem steruje przepływem danych do pierwszej instrukcji Podprogramu. pierwsze dwie instrukcje push odkładają rejestry ax i bx na stos, przypuszczalnie żeby zachować ich wartości ponieważ Podprogram je modyfikuje. Niestety ,istnieje błąd programistyczny w powyższym programie, Podprogram tylko zdejmuje bx ze stosu ,ale zapomina zdjąć również ax. To znaczy, że kiedy podprogram próbuje wrócić do wywołania, wartość ax prędzej niż inne zwróci adres powrotu, który siedzi na szczycie stosu. Dlatego też, ten Podprogram zwraca sterowanie do adresu wyszczególnionego przez wartość początkową rejestru ax zamiast prawdziwego adresu powrotu .Ponieważ jest 65,536 różnych wartości jakie może mieć ax, jest jedna szansa na 65,536 ,że nasz kod powróci do prawdziwego adresu powrotu, Bardziej prawdopodobne, że kod taki jak ten zawiesi naszą maszynę. Morał z tej historii – zawsze upewniaj się, że adres powrotu jest usadowiony na stosie przed wykonaniem instrukcji powrotu.

Podobnie jak instrukcja call, jest bardzo łatwo do symulowania instrukcji ret używając dwóch instrukcji 80x86.Wszystko co musimy zrobić to zdjąć adres powrotu ze stosu a potem skopiować go do rejestru ip .Dla bliskiego powrotu, jest to bardzo prosta operacja, zdejmujemy bliski adres powrotu ze stosu a potem skaczemy pośrednio do tego rejestru.:

```
pop     ax
jmp     ax
```

Symulacja dalekiego powrotu jest trochę bardziej trudniejsza ponieważ musimy załadować cs:ip w pojedynczej operacji. Jedyną instrukcją która to robi jest instrukcja jmp mem_{32}.

Są dwie różne formy instrukcji ret. Są one identyczne do tych powyższych z wyjątkiem 16 bitowego przesunięcia ich opcodów. CPU dodaje te wartości do wskaźnika stosu bezpośrednio po zdjęciu adresu powrotu ze stosu. Mechanizm ten usuwa parametry włożone na stos przed powrotem z wywołania.

Asembler pozwala nam pisać bez przyrostków „f” lub „n” .jeśli to zrobimy, asembler wykombinuje czy powinien wygenerować bliski czy daleki powrót.

6.9.3 INSTRUKCJE INT,INTO,BOUND I IRET

Instrukcja int (wywołanie programu obsługi przerwania) jest bardzo specjalną formą instrukcji call. Podczas gdy instrukcja call wywołuje podprogram wewnątrz naszego programu, instrukcja int wywołuje system podprogramów i inne specjalne podprogramy. Główna różnica między podprogramem obsługi przerwań a standardowymi procedurami jest taka ,że możemy mieć każdą liczbę różnych procedur w programie asemblerowym, podczas gdy system wspiera maksimum 256 różnych podprogramów obsługi przerwań. Program wywołuje podprogram poprzez wyszczególnienie adresu tego podprogramu; wywołuje podprogram obsługi przerwań przez wyszczególnienie numeru przerwania dla poszczególnego podprogramu obsługi przerwań. ten rozdział omawia tylko jak wywołać podprogram obsługi przerwań, używając instrukcji int, into i bound, i jak powrócić z podprogramu obsługi przerwań używając instrukcji iret.

Są cztery różne formy instrukcji int. Pierwszą formą jest

```
int     nm
```

(gdzie „nm" jest wartością między 0 a 256).Pozwala ona nam wywołać jedno z 256 różnych podprogramów przerwań. Forma ta instrukcji int jest długa na dwa bajty. Pierwszym bajtem jest opcod int. Drugim bajtem jest dana bezpośrednia zawierająca numer przerwania.

Chociaż możemy używać instrukcji int do wywołania procedur (podprogram obsługi przerwań),pierwszym celem tej instrukcji jest wywołanie funkcji systemowej. Funkcja systemowa jest procedurą dostarczaną przez system taki jak DOS, mysz PC-BIOS lub kilka innych programów rezydujących w maszynie przed tym zanim program zacznie się wykonywać. Ponieważ zawsze odwołujemy się do wyszczególnionej funkcji systemowej przez jej numer przerwania zamiast adres, nasz program nie musi znać aktualnego adresu podprogramu w pamięci. Instrukcja int dostarcza łączenia dynamicznego do naszego programu. CPU określa aktualny adres podprogramu obsługi przerwań w czasie wykonania poprzez szukanie tego adresu w tablicy wektora przerwań. Pozwala to autorowi takiego systemu podprogramów do zmiany kodu (wliczając w to punkt wejścia) bez obawy o starsze programy które wywołują podprogram obsługi przerwań .Tak długo jak funkcja systemowa używa tego samego numeru przerwania, CPU automatycznie wywołuje podprogram obsługi przerwań spod nowego adresu.

Jedyny problem z instrukcją int jest taki, że wspiera tylko 256 różnych podprogramów obsługi przerwań. MS-DOS sam używa ponad 100 różnych wywołań. BIOS i inne oprogramowanie systemowe dostarcza tysiące innych. Te i wszelkie inne wszystkie przerwania zarezerwowane przez Intel dla przerwań sprzętowych i pułapek. Powszechnym rozwiązaniem w większości funkcji systemowych jest stosowanie pojedynczego numeru przerwania dla danej klasy wywołania a potem podanie numery funkcji w jednym z rejestrów 80x86 (typowo rejestrze ax).Na przykład, MS-DOS używa tylko pojedynczego numeru przerwania, 21h.Wybierając szczególną funkcję DOS, ładujemy kod funkcji DOS do rejestru ah przed wykonaniem instrukcji int 21h.Na przykład, przerwanie programu i przywrócenie sterowania do MS-DOS wymaga załadowania 4Ch i wywołania DOSa instrukcją int 21h:

```
mov     ah,4ch
int     21h
```

Przerwanie klawiatury BIOS jest dobrym przykładem. Przerwanie 16h jest odpowiedzialne za testowanie i odczytywanie danych z klawiatury .ten podprogram BIOS dostarcza kilku wywołań do odczytu znaków i kodu klawisza klawiatury, aby zobaczyć czy jakieś klawisze są dostępne w systemowym buforze, sprawdza stan klawiatury modyfikując flagi, i wiele innych. Wybierając poszczególne operacje ładujemy numer funkcji do rejestru ah przed wykonaniem int 16h.Następująca tabela wymienia możliwe funkcje:

Function # (AH)	Input Parameters	Output Parameters	Description
0		al- ASCII character ah- scan code	Read character. Reads next available character from the system's type ahead buffer. Wait for a keystroke if the buffer is empty.
1		ZF- Set if no key. ZF- Clear if key available. al- ASCII code ah- scan code	Checks to see if a character is available in the type ahead buffer. Sets the zero flag if not key is available, clears the zero flag if a key is available. If there is an available key, this function returns the ASCII and scan code value in ax. The value in ax is undefined if no key is available.
2		al- shift flags	Returns the current status of the shift flags in al. The shift flags are defined as follows: bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Alt key is down bit 2: Ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down
3	al = 5 bh = 0, 1, 2, 3 for 1/4, 1/2, 3/4, or 1 second delay bl= 0..1Fh for 30/sec to 2/sec.		Set auto repeat rate. The bh register contains the amount of time to wait before starting the autorepeat operation, the bl register contains the autorepeat rate.
5	ch = scan code cl = ASCII code		Store keycode in buffer. This function stores the value in the cx register at the end of the type ahead buffer. Note that the scan code in ch doesn't have to correspond to the ASCII code appearing in cl. This routine will simply insert the data you provide into the system type ahead buffer.
10h		al- ASCII character ah- scan code	Read extended character. Like ah=0 call, except this one passes all key codes, the ah=0 call throws away codes that are not PC/XT compatible.
11h		ZF- Set if no key. ZF- Clear if key available. al- ASCII code ah- scan code	Like the ah=01h call except this one does not throw away keycodes that are not PC/XT compatible (i.e., the extra keys found on the 101 key keyboard).

Function # (AH)	Input Parameters	Output Parameters	Description
12h		al- shift flags ah- extended shift flags	Returns the current status of the shift flags in ax. The shift flags are defined as follows: bit 15: SysReq key pressed bit 14: Capslock key currently down bit 13: Numlock key currently down bit 12: Scroll lock key currently down bit 11: Right alt key is down bit 10:Right ctrl key is down bit 9: Left alt key is down bit 8: Left ctrl key is down bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Either alt key is down (some machines, left only) bit 2: Either ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down

Tabela 31: Funkcje wspierające klawiaturę BIOS

Na przykład odczytując znak z bufora systemowego pozostawiamy kod ASCII w al., możemy użyć kodu:

```
mov     ah, 0           ;czekanie na dostępny kod ,a potem
int     16h             ;odczyt tego klawisza
mov     znak, al.       ;zachowanie odczytanego znaku
```

Podobnie, jeśli chcielibyśmy przetestować bufor aby zobaczyć czy klawisz jest dostępny bez odczytywania naciśniętego klawisza, możemy użyć następującego kodu:

```
mov     ax, 1           ;testujemy aby zobaczyć czy klawisz jest dostępny
int     16h             ;ustawiamy flagę zera jeśli klawisz nie jest dostępny
```

Po więcej informacji o PC-BIOS i MS-DOS zobacz w „MS-DOS,PC-BIOS i pliki I/O”

Drugą formą instrukcji int jest specjalny przypadek:

```
Int     3
```

Int 3 jest specjalną formą instrukcji przerwania, która jest długości jednego bajta. CodeView i inne debuggery używają jej jako instrukcji programowego przerwania. Kiedykolwiek ustawimy punkt przerwania na instrukcji w naszym programie, debugger zastąpi pierwszy bajt opcodu instrukcji instrukcją int 3.Kiedy nasz program wykona instrukcję int 3,odwoła się do „funkcji systemowej’ debuggera, więc debugger odbierze sterowanie z CPU. Kiedy się to stanie, debugger zastąpi instrukcję int 3 oryginalnym opcodem.

Kiedy działamy wewnątrz debuggera, możemy używać instrukcji int 3 do zatrzymania wykonywania programu i zwrócenia sterowania do debuggera. Nie jest to normalny sposób zatrzymywania programu.. Jeśli spróbujemy wykonać instrukcję int 3 podczas działania pod DOSem, zamiast pod kontrolą debuggera, możemy łatwo załatwić system.

Trzecią formą instrukcji int jest into .Into będzie powodowało programowe przerwanie jeśli flag przepełnienia będzie ustawiona. Możemy użyć tej instrukcji do szybkiego testowania dla arytmetycznego przepełnienia po wykonaniu instrukcji arytmetycznych .Semantycznie instrukcja ta jest odpowiednikiem:

```
if   overflow = 1  then int 4
```

Nie powinniśmy używać tej instrukcji jeżeli nie dostarczymy odpowiedniej procedury pułapki (podprogram obsługi przerwań).Robiąc to spowodujemy prawdopodobnie zawieszenia systemu.

Czwartym przerwaniem programowym dostarczonym przez procesory 80286 i późniejsze, jest instrukcja bound. Instrukcja ta przyjmuje formę:

```
bound    reg, mem
```

i wykonuje następujący algorytm:

```
if  (reg < [mem]) or (reg > [mem+sizeof(reg)]) then int 5
```

[mem] oznacza zawartość komórki pamięci a sizeof (reg0 to dwa lub cztery w zależności od tego czy rejestr jest szeroki na 16 czy 32 bity. Operand pamięci musi mieć podwójny rozmiar w stosunku do operandu rejestru. Instrukcja bound porównuje wartości używając całkowitego porównania ze znakiem.

Projektanci Intela dodali instrukcję bound pozwalając szybko sprawdzić zakres wartości w rejestrze. Jest to użyteczne w Pascalu, który sprawdza zakres tablicy a po sprawdzeniu sprawdza czy ciąg liczb integer jest w poprawnej granicy. Są dwa problemy z tą instrukcją. Na procesorach 80486 i Pentium/586 instrukcja bound jest generalnie wolniejsza niż instrukcji które podmienia:

```
cmp     reg, LowerBound
jl      OutOfBound
cmp     reg, UpperBound
jg      OutOfBound
```

Na chipach 80486 i Pentium/586,powyższa sekwencja wymaga tylko czterech cykli zegarowych ,zakładając ,że możemy użyć bezpośredniego trybu adresowania i rozgałęzienia nie są pobierane; instrukcja bound wymaga 7-8 cykli zegarowych w podobnych okolicznościach i również zakładając, że operandy pamięci są w pamięci podręcznej.

Drugi problem z instrukcją bound jest taki, że wykonuje się jako int 5 jeśli wyszczególniony rejestr jest poza zakresem. IBM, w swojej nieskończonej mądrości, postanowił używać przerwania int 5 do drukowania ekranu. Dlatego też jeśli wykonujemy instrukcję bound i wartość jest poza zakresem, system, domyślnie ,zacznie drukować zawartość ekranu na drukarce. Jeśli zamienimy procedurę int 5 na jeden z naszych w własnych, naciskając klawisz PrtSc przekażemy sterowanie do podprogramu naszej instrukcji bound. Chociaż są sposoby ominięcia tego problemu, większość ludzi nie martwi się tym ponieważ instrukcja bound jest zbyt wolna.

Jakąkolwiek instrukcję int wykonujemy, wystąpi następująca sekwencja zdarzeń:

- 80x86 odkłada rejestr flag na stos
- 80x86 odkłada cs a potem ip na stos
- 80x86 używa numeru przerwania (into jest przerwaniem #4,bound #5) razy cztery jako indeks do tablicy wektora przerwań i kopiuje podwójne słowo z punktu w tablicy do cs:ip.

Instrukcje int różnią się od call w dwóch głównych punktach .Po pierwsze, instrukcje call różnią się w długości od dwóch do sześciu bajtów długości ,podczas gdy instrukcje int są generalnie długie na dwa bajty (int 3,into i bound są wyjątkami)Po drugie, i bardzo ważne, instrukcja int odkłada flagi i adres powrotu na stos podczas gdy instrukcja call odkłada tylko adres powrotu. Zauważmy również, że instrukcja int zawsze odkłada daleki adres powrotu (tj. wartość cs i offset wewnątrz segmentu kodu),gdy dalekie wywołanie odkłada podwójne słowo adresu powrotu.

Ponieważ int odkłada flagi na stos musimy użyć specjalnej instrukcji powrotu, iret (powrót z przerwania),powrót z podprogramu wywołującego przez instrukcję int. Jeśli wracamy z procedury przerwania używając instrukcji ret ,flagi pozostaną na stosie by powrócić do tego wywołania który je przywołał .Instrukcja iret jest odpowiednikiem sekwencji dwóch instrukcji: ret, popf(zakładając, że wykonaliśmy popf przed zwróceniem sterowania pod adres wskazywany przez podwójne słowo na szczycie stosu).

Instrukcje int czyszczą flagę śledzenia (T) w rejestrze flag. nie wpływają na żadną inna flagę. Instrukcja iret, przez wzgląd na swoją naturę, może wpływać na wszystkie flagi ponieważ zdejmuje flagi ze stosu.

6.9.4 INSTRUKCJE SKOKÓW WARUNKOWYCH

Chociaż instrukcje jmp, call i ret dostarczają sterowania przepływem danych, nie pozwalają nam na podjęcie poważnych decyzji. Instrukcje skoków warunkowych wykonują to zadanie .Instrukcje skoków warunkowych są podstawowymi narzędziami dla tworzenia pętli i innych warunkowo wykonywanych instrukcji takich jak if...then.

Skoki warunkowe testują jedną z wielu flag w rejestrze flag aby zobaczyć czy pasują do wzorca (podobnie jak instrukcje setcc).Jeśli wzór pasuje, sterownie przepływem danych jest przekazywane do lokacji docelowej .Jeśli wzór nie odpowiada, CPU ignoruje skok warunkowy i kontynuuje wykonywanie od następnej instrukcji. Niektóre instrukcje, testują warunki flag znaku, przepełnienia, przeniesienia i zera. Na przykład, po wykonaniu instrukcji przesunięcia w lewo, możemy sprawdzić flagę przeniesienia aby ustalić czy przeniesiono jeden poza bardziej znaczący bit operandu. podobnie, możemy sprawdzić warunek flagi zera po instrukcji test aby zobaczyć czy wyszczególniony bit był jedynką. Większość czasu jednak, będziemy prawdopodobnie wykonywać skoki warunkowe po instrukcji cmp. Instrukcja cmp ustawia flagi ,więc możemy sprawdzić dla mniejsze niż, większe niż, równe itp.

Notka: Dokumentacja Intelowska definiuje kilka synonimów lub aliasów instrukcji dla wielu instrukcji skoków warunkowych. Następujące tablice wyliczają wszystkie aliasy dla poszczególnych instrukcji. Tablice te również wyliczają skoki przeciwne Zobaczymy wkrótce cel przeciwnych rozgałęzień.

Instruction	Description	Condition	Aliases	Opposite
JC	Jump if carry	Carry = 1	JB, JNAE	JNC
JNC	Jump if no carry	Carry = 0	JNB, JAE	JC
JZ	Jump if zero	Zero = 1	JE	JNZ
JNZ	Jump if not zero	Zero = 0	JNE	JZ
JS	Jump if sign	Sign = 1		JNS
JNS	Jump if no sign	Sign = 0		JS
JO	Jump if overflow	Ovrflw=1		JNO
JNO	Jump if no Ovrflw	Ovrflw=0		JO
JP	Jump if parity	Parity = 1	JPE	JNP
JPE	Jump if parity even	Parity = 1	JP	JPO
JNP	Jump if no parity	Parity = 0	JPO	JP
JPO	Jump if parity odd	Parity = 0	JNP	JPE

Tablica 32: Instrukcje Jcc, które testują flagi

Instruction	Description	Condition	Aliases	Opposite
JA	Jump if above (>)	Carry=0, Zero=0	JNBE	JNA
JNBE	Jump if not below or equal (not <=)	Carry=0, Zero=0	JA	JBE
JAE	Jump if above or equal (>=)	Carry = 0	JNC, JNB	JNAE
JNB	Jump if not below (not <)	Carry = 0	JNC, JAE	JB
JB	Jump if below (<)	Carry = 1	JC, JNAE	JNB
JNAE	Jump if not above or equal (not >=)	Carry = 1	JC, JB	JAE
JBE	Jump if below or equal (<=)	Carry = 1 or Zero = 1	JNA	JNBE
JNA	Jump if not above (not >)	Carry = 1 or Zero = 1	JBE	JA
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

Tablica 33:Instrukcje Jcc dla porównań bez znakowych

Instruction	Description	Condition	Aliases	Opposite
JG	Jump if greater (>)	Sign = Ovrflw or Zero=0	JNLE	JNG
JNLE	Jump if not less than or equal (not <=)	Sign = Ovrflw or Zero=0	JG	JLE
JGE	Jump if greater than or equal (>=)	Sign = Ovrflw	JNL	JGE
JNL	Jump if not less than (not <)	Sign = Ovrflw	JGE	JL
JL	Jump if less than (<)	Sign ≠ Ovrflw	JNGE	JNL
JNGE	Jump if not greater or equal (not >=)	Sign ≠ Ovrflw	JL	JGE
JLE	Jump if less than or equal (<=)	Sign ≠ Ovrflw or Zero = 1	JNG	JNLE
JNG	Jump if not greater than (not >)	Sign ≠ Ovrflw or Zero = 1	JLE	JG
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

Tablica 34 Instrukcje Jcc dla porównań ze znakiem

W procesorach 80286 i wcześniejszych, instrukcje te wszystkie są dwubajtowe. Pierwszy bajt jest jednym bajtem opcodu następujący po jednym bajcie przesunięcia. Chociaż to prowadzi do instrukcji o bardzo małych wymiarach, pojedynczy bajt przesunięcia pozwala tylko na zakres +/- 138 bajtów .Jest prosta sztuczka, którą możemy zastosować aby przezwyciężyć to ograniczenie na wcześniejszych procesorach:

- Jakikolwiek skok użyjemy ,zamieniamy na formę przeciwną (podaną w powyższych tabelach)
- Jeśli wybraliśmy skok przeciwny, używamy instrukcji jmp której adres docelowy jest oryginalnym adresem docelowym.

Na przykład konwersja:

```
        jc      Cel
```

używa sekwencji instrukcji:

```
        jnc     SkiJmp
        jmp     Cel
SkiJmp:
```

Jeśli flaga przeniesienia jest czysta (NC= no carry – żadnego przeniesienia),wtedy sterowanie jest przenoszone do etykiety SkiJmp, do tego samego miejsca, gdybyśmy użyli instrukcji jc. Jeśli flaga przeniesienia jest ustawiona kiedy napotykamy tą sekwencję, sterownie zostanie przekazane poprzez instrukcję jnc do instrukcji jmp ,która przeniesie sterowanie do celu .Ponieważ instrukcja jmp pozwala na 16 bitowe przesunięcie i daleki operand, możemy skoczyć gdziekolwiek w pamięci używając tej sztuczki,

Krótki komentarz do kolumny „przeciwne" Jak wspomniano powyżej, kiedy musimy ręcznie rozszerzyć rozgałęzienie z +/- 128 powinniśmy wybrać rozgałęzienie przeciwne by móc wykonać skok do lokacji docelowej. Jak już widzieliśmy w kolumnie „aliasy" ,wiele instrukcji skoków warunkowych ma aliasy. To znaczy, że będą również aliasy dla skoków przeciwnych. Nie używamy żadnych aliasów kiedy rozszerzamy rozgałęzienia które są poza zakresem.Za wyjątkiem dwóch wyjątków, bardzo prosta zasada opisuje jak wygenerować rozgałęzienie przeciwne:

- Jeśli druga litera instrukcji jcc nie jest „n", wprowadzamy „n" po „j". Np. je staje się jne a jl jnl
- Jeśli druga litera instrukcji jcc jest „n" wtedy usuwamy to „n" z instrukcji. Np. jng staje się jg a jne je

Te dwa wyjątki od tej zasady to jpe (skok przy parzystości) i jpo (skok przy braku parzystości).Wyjątki te powodują kilka problemów ponieważ (a) prawie zawsze musimy sprawdzać flagę parzystości (b) możemy użyć aliasów jp i jnp jao synonimów dla jpe i jpo. Zasada „N /No N" stosuje się do jp i jnp.

Mimo, że wiemy iż jge jest przeciwne jl, miejmy w zwyczaju używanie jnl zamiast jge. Jest zbyt łatwo w ważnej sytuacji zacząć myśleć „większe jest w opozycji do mniejsze" i zastąpić jg. Możemy uniknąć tej pomyłki zawsze używając zasady „N / No N"

MASM 6x i wiele innych nowoczesnych asemblerów 80x86 automatycznie konwertuje poza zakresem rozgałęzienia dla tej sekwencji. Jest opcja która pozwala nam zablokować tą cechę. Dla wykonania krytycznego kodu pracującego na 80286 i wcześniejszych procesorach ,możemy chcieć zablokować tą cechę więc możemy

poprawić własne rozgałęzienie. Powód jest bardzo prosty, ta prosta poprawka zawsze uniknie potoku niezależnie jest prawdziwa ponieważ CPU skacze w obu przypadkach. Jedna miła rzecz skoków warunkowych jest taka, że nie opróżniamy potoku i kolejki rozkazów jeśli nie wykonujemy rozgałęzienia. Jeśli jeden warunek jest prawdziwy dużo bardziej niż inny, możemy chcieć użyć skoku warunkowego do przeniesienia sterowania do pobliskiego jmp ,więc możemy kontynuować tak jak przedtem Na przykład, jeśli mamy instrukcję je cel i cel jest poza zakresem ,możemy skonwertować ja na następujący kod:

```
                je      GotoTarget
                -
                -
                -
GotoTarget:     jmp     Cel
```

Chociaż rozgałęzienie do celu wymaga teraz dwóch skoków, jest to bardziej wydajne niż standardowa konwersja jeśli flaga zera jest normalnie wyczyszczona kiedy wykonujemy instrukcję je.

Procesor 80386 i późniejsze, dostarcza rozszerzonej formy skoku warunkowego który jest czterobajtowy ,z ostatnimi dwoma bajtami zawierającymi 16 bitowe przesunięcie. Te skoki warunkowe mogą przekazywać sterowanie gdziekolwiek wewnątrz bieżącego segmentu kodu. Dlatego też, nie musimy się martwić o ręczne rozszerzanie zakresu skoku. Jeśli powiemy MASMowi, że używamy 80386 lub późniejszych procesorów, on automatycznie wybierze dwa lub cztery bajty jeśli będzie to konieczne. Zobacz Rozdział Ósmy, uczący jak powiedzieć MASMowi ,że używamy procesora 80386 lub późniejszych.

Instrukcje skoków warunkowych 80x86 dają nam możliwość podziału przebiegu programu na jedną z dwóch części w zależności kilku warunków logicznych. Przypuśćmy, że chcemy zwiększyć rejestr ax jeśli bx jest równy cx. Możemy osiągnąć to za pomocą następującego kodu:

```
                cmp     bx, cx
                jne     SkipStmts
                inc ax
SkipStmts:
```

Sztuczką jest użycie rozgałęzienia „przeciwnego” aby przeskoczyć nad instrukcją którą chcemy wykonać jeśli warunek jest prawdziwy. Zawsze używamy zasady „rozgałęzienia przeciwnego (N /no N)” podanej wcześniej do wyboru rozgałęzienia przeciwnego. Możemy zrobić ten sam błąd wybierając rozgałęzienie przeciwne kiedy mogliśmy to zrobić poza zakresem skoków.

Możemy również użyć instrukcji skoków warunkowych do połączenia pętli. Na przykład, następująca sekwencja kodu odczytuje sekwencję znaków od użytkownika i przechowuje każdy znak w kolejnych elementach tablicy do chwili kiedy użytkownik naciśnie klawisz ENTER (powrót karetki):

```
                mov     di, 0
ReadLnLoop:     mov     ah, 0           ;INT  16h czyta opcod klawisza
                int     16h
                mov     Input[di], al.
                inc     di
                cmp     al., 0dh        ;kod ASCII powrotu karetki
                jne     ReadLnLoop
                mov     Input [di-1],0
```

Po więcej informacji dotyczącej połączenia instrukcji IF, pętli i innych struktur sterujących zobacz „Struktury Sterujące”

Podobnie jak instrukcje setcc, instrukcje skoków warunkowych dzielą się na dwie podstawowe kategorie – te które testują wyszczególnione wartości flag (np. jz, jc, jno) i te ,które testują inne warunki(mniejszy niż, większy niż itp.).kiedy sprawdzamy warunek, instrukcje skoków warunkowych prawie zawsze występują po instrukcji cmp. Instrukcja cmp ustawia flagi, więc możemy używać instrukcji ja, jae, jb, jbe, je lub jne do sprawdzania bez znakowego mniejszego niż, mniejszego lub równego, równości, nierówności, większego niż lub większego niż lub równego. Równocześnie instrukcja cmp ustawia flagi więc możemy również zrobić porównania ze znakiem używając instrukcji jl, jle, je, jne, jg i jge.

Instrukcje skoków warunkowych tylko sprawdzają flagi, nie wpływają na żadną z flag 80x86.

6.9.5 INSTRUKCJE JCXZ/JECXZ

Instrukcja jcxz(skocz jeśli cx wynosi zero) rozgałęzia do adresu docelowego jeśli cx zawiera zero. Chociaż możemy użyć jej zawsze musimy wiedzieć, że czy cx zawiera zero, moglibyśmy normalnie użyć jej przed pętlą skonstruowaną w oparciu o instrukcję loop. Instrukcja loop może powtarzać sekwencję operacji cx razy .Jeśli cx równa się zero, loop będzie powtarzał operację 65,536 razy. Możemy użyć jcxz do przeskoczenia takiej pętli kiedy cx wynosi zero.

Instrukcja jecxz, dostępna tylko na 80386 i późniejszych procesorach, robi zasadniczo taką samą prace jak jcxz ,z wyjątkiem tego, że testuje pełny rejestr ecx. Zauważmy ,że instrukcja jcxz sprawdza tylko cx ,nawet na 80386 w 32 bitowym trybie.

Nie ma „przeciwnych" instrukcji do jcxz i jecxz. Dlatego też ,nie możemy użyć zasady „N /No N" do rozszerzenia instrukcji jcxz i jecxz. Najłatwiejszym sposobem rozwiązania tego problemu jest rozbicie instrukcji na dwie instrukcje które wykonują to samo zadanie:

```
        jcxz    Cel
        -
        test    cx, cx          ;ustawienie flagi zera jeśli cx = 0
        je      Cel
```

teraz możemy już łatwo rozszerzyć instrukcję je używając techniki z poprzedniej sekcji.

Instrukcja test ustawia flagę zera jeśli i tylko jeśli cx zawiera zero. W końcu, jeśli są jakieś niezerowe bity w cx, logiczne dodanie ich z nimi samymi stworzy nie zerowy wynik. Jest to doby sposób aby sprawdzić czy 16 lub 32 bitowy rejestr zawiera zero. Faktycznie ta sekwencja tych instrukcji jest szybsza niż instrukcja jcxz na 80486 i późniejszych procesorów. Intel rekomenduje użycie tej sekwencji zamiast instrukcji jcxz jeśli zależy nam na szybkości .Oczywiście, instrukcja jcxz jest krótsza niż sekwencja dwóch instrukcji, ale nie jest szybsza. Jest to dobry przykład wyjątku od zasady „krótsze jest zazwyczaj szybsze".

Instrukcja jcxz nie wpływa na żadną z flag.

6.9.6 INSTRUKCJA LOOP

Ta instrukcja zmniejsza rejestr cx a potem rozgałęzia do lokacji docelowej jeśli rejestr cx nie zawiera zera. Ponieważ instrukcja ta zmniejsza cx, wtedy sprawdza dla zera czy cx początkowo zawierał zero, każda pętla jaką stworzymy używa instrukcji loop 65,536 razy. Jeśli nie chcemy wykonywać pętli kiedy cx zawiera zero ,użyjemy jcxz do przeskoczenia pętli.

Nie ma „przeciwnych" form instrukcji loop, i podobnie jak instrukcje jcxz /jecxz zakres jej jest ograniczony do +/- 128 bajtów na wszystkich procesorach. Jeśli chcemy rozszerzyć zakres tej instrukcji, będziemy musieli ją rozbić:

;"loop lbl":

```
        dec     cx
        jne     lbl
```

Możemy łatwo rozszerzyć jne na każdą odległość.

Nie ma instrukcji loop która zmniejsza ecx i rozdziela jeśli nie zero (jest instrukcja loope, ale robi ona całkowicie coś innego).Powód jest całkiem prosty. Wraz z 80386,projektanci Intela całkowicie wstrzymali rozwój instrukcji loop. Oh, jest ona dla zapewnienia zgodności ze starszym kodem ,ale okazuje się, że instrukcje dec /jne są rzeczywiście szybsze na procesorach 32 bitowych. Problemy w dekodowaniu instrukcji i działaniach potoku odpowiadają za ten dziwny zbieg zdarzeń.

Chociaż nazwa instrukcji loop sugeruje, że możemy normalnie tworzyć z nią pętle, musimy pamiętać ,że wszystko co rzeczywiście możemy z nią zrobić to zmniejszanie cx i rozgałęzienie do adresu docelowego ,jeśli cx nie zawiera zera po zmniejszeniu. Możemy użyć tej instrukcji gdzie chcemy zmniejszyć cx a potem sprawdzić wynik dla zera .Pomimo to, jest to bardzo dogodna instrukcja do używania jeśli po prostu chcemy powtórzyć sekwencję instrukcji jakąś ilość razy. Na przykład. następująca inicjuje 256 elementów tablicy bajtów wartościami1,2,3...

```
        mov     ecx, 255
ArrayLp:mov     Array[ecx], sl
        loop    ArrayLp
        mov     Array[0], 0
```

Ostania instrukcja jest konieczna ponieważ pętla nie powtarza się kiedy cx wynosi zero. Dlatego też, w związku z ostatnią instrukcją ,ostatnim elementem tablicy na której działa loop jest Array[1].

Instrukcja loop nie wpływa na żadną z flag.

6.9.7 INSTRUKCJE LOOPE/LOOPZ

Loope /loopz rozgałęziają do adresu docelowego jeśli cx nie jest równe zero a flaga zera jest ustawiona. Instrukcja ta jest całkiem użyteczna po instrukcjach cmp i cmps, i jest odrobinę szybsza niż porównywalne instrukcje 803886/496 jeśli używamy wszystkich cech tej instrukcji .Jednakże instrukcja ta dokonuje spustoszenia w potoku i operacjach superskalarnych Pentium, więc prawdopodobnie przyzwyczaić się do instrukcji dyskretnych zamiast tej instrukcji. Instrukcja ta robi co następuje:

```
cx := cx -1
ix Flaga_Zera = 1 and cx ≠ 0 , goto cel
```

Instrukcja loope zawodzi jeśli występuje jeden z dwóch warunków. Albo flaga zera jest wyzerowana albo instrukcja zmniejszyła cx do zera.. Poprzez testowanie flagi zera po instrukcji loop (instrukcją je lub jne na przykład),możemy określamy przyczynę zatrzymania.

Instrukcja ta jest użyteczna jeśli musimy powtórzyć pętle jeśli jakaś wartość jest równa innej, ale jest maksymalna liczba iteracji na jaką możemy pozwolić. Na przykład następująca pętla przeszukuje tablicę szukając pierwszego nie zerowego bajtu, ale nie szuka poza końcem tablicy:

```
                mov     cx, 16          ;max. 16 elementów tablicy
                mov     bx, -1          ;indeks do tablicy (następne inc)
SerachLp:       inc     bx              ;przesuwa się na następny element tablicy
                cmp     Array[bx], 0    ;zobacz czy ten element to zero
                loope   SearchLp        ;powtórz jeśli jest
                je      AllZero         ;skocz jeśli wszystkie elementy były zerami
```

Zauważmy, że instrukcja ta nie jest przeciwieństwem loopnz / loopne .Jeśli musimy rozszerzyć ten skok poza +/- 128 bajtów, będziemy musieli połączyć tą instrukcję używając instrukcji dyskretnych. Na przykład, jeśli cel loope jest poza zakresem, będziemy musieli użyć sekwencji instrukcji podobnej do tej:

```
                jne     quite
                dec     cx
                je      quite2
                jmp     cel
quite:          dec     cx              ; loope zmniejsza cx, nawet jeśli ZF = 0
quite2:
```

Instrukcja loope /loopz nie wpływają na żadną flagę.

6.9.8 INSTRUKCJA LOOPNE/LOOPNZ

Instrukcja ta jest podobna do instrukcji loope/ loopz z poprzedniej sekcji z wyjątkiem loopne /loopnz powtarza gdy cx nie jest zerem i flaga zera jest wyczyszczona. Algorytm:

```
cx := cx – 1
if Flaga_Zera = 0 and cx ≠ 0 , goto cel
```

Możemy określić czy instrukcja loopne zakończy ponieważ cx było zerem lub czy flaga zera była ustawiona przez testowanie flagi zera bezpośrednio po instrukcji loopne. Jeśli flaga zera jest wyczyszczona w tym momencie, instrukcja loopne nie dochodzi do skutku ponieważ zmniejsza ona cx do zera. W przeciwnym razie nie dochodzi do skutku, ponieważ flaga zera była ustawiona.

Instrukcja ta nie jest przeciwieństwem instrukcji loope / loopz. Jeśli adres docelowy jest poza zakresem, będziemy musieli użyć sekwencji instrukcji podobnej do tej:

```
                je      quit
                dec     cx
                je      Quit2
                jmp     Cel
quit:           dec     cx              ;loopne zmniejsza cx ,nawet jeśli ZF = 1
quit2:
```

Możemy użyć instrukcji loopne do powtarzania określonej ilości skoków podczas czekania na jeden warunek by był prawdziwy **.**Na przykład, możemy przeszukać tablicę aż do wyczerpania liczby elementów tablicy lub aż do znalezienia pewnego bajtu używając pętli jak następuje:

```
                mov     cx, 16          ;Maksimum elementów tablicy
                mv      bx, -1          ;indeks do tablicy
LoopWhlNot0:    inc     bx              ;Przesunięcie na następny element
                cmp     Array[bx], 0    ;czy ten element zawiera zero?
                loopne  LoopWhlNot0     ;wyjście jeśli tak lub więcej niż 16 bajtów
```

Chociaż instrukcja loope/ loopz i loopne/ loopnz są wolniejsze niż pojedyncze instrukcje z których mogłyby być połączone**,** jest jedno główne zastosowanie dla tych form instrukcji gdzie szybkość nie jest tak ważna: bycie szybszym powoduje ich mniejszą użyteczność – pętle z licznikiem podczas operacji I/O. Przypuśćmy ,że bit #7 na wejściu portu 379h zawiera jeden jeśli urządzenie jest zajęte i zawiera zero jeśli urządzenie nie jest zajęte. Jeśli chcemy wysłać dane do portu możemy użyć następującego kodu:

```
                mov     dx, 379h
WaitNoBusy:     in      al., dx         ;pobiera port
                test    al., 80h        ;patrzy czy bit #7 jest jedynką
                jne     WaitNoBusy      ;Czekaj jeśli :nie zajęty"
```

Jedyny problem z tą pętlą jest taki, że jest niewykluczone iż będzie to pętla nieskończona. W prawdziwym systemie, przewód może być wyłączony, ktoś mógł wyłączyć urządzenie peryferyjne, i wiele innych rzeczy może się wydarzyć złych, które spowodują zawieszenie systemu. Dobry program zwykle nakłada

limit czasowy na pętlę taka jak ta. Jeśli urządzenie wydaje się być zajęte przez określoną ilość czasu, wtedy pętla kończy się i wzbudza warunek błędu .Realizuje to następujący kod:

```
                mov     dx, 379h        ;adres portu wejściowego
                mov     cx, 0           ;Pętla 65,536 razy a potem wyjście
WaitNo Busy:    in      al., dx         ;Pobieranie danych z portu
                test    al., 80h        ;zobacz czy zajęty
                loopne  WaitNoBusy      ;powtórz  jeśli zajęty i licznik nie wyzerowany
                jne     TimeOut         ;rozgałęzienie jeśli CX=0 ponieważ skończył się czas
```

Możemy użyć instrukcji loope/ loopz jeśli bit był zerem a nie jedynką.

Instrukcja loopne /loopnz nie wpływają na żadną flagę

6.10 INSTRUKCJE RÓŻNORODNE

Jest kilka różnorodnych instrukcji w 80x86 które nie podpadają pod żadną z wyżej wymienionych kategorii .generalnie są to instrukcje które manipulują pojedynczymi flagami ,dostarczają specjalnej obsługi procesora lub operują operacjami uprzywilejowanego trybu.

Jest kilka instrukcji które bezpośrednio manipulują flagami w rejestrze flag 80x86.

Są to:

* clc	Wyzerowanie flagi przeniesienia
* stc	Ustawienie flagi przeniesienia
* cmc	Zamiana wartości flagi przeniesienia na przeciwną
* cld	Wyzerowanie flagi kierunku
* std	Ustawienie flagi kierunku
* cli	Wyzerowanie flagi zezwolenia na przerwanie
* sti	Ustawienie flagi zezwolenia na przerwanie

Notka: powinniśmy być ostrożni kiedy używamy instrukcji cli w naszych programach .Niewłaściwe zastosowanie może zawiesić maszynę do momentu wyłączenia z prądu

Instrukcja nop nie robi nic za wyjątkiem marnowania kilu cykli procesora i zajmowania bajtu pamięci. Programiści często używają jej jako wypełniacz lub wspomaganie debuggowania .Okazuje się, że nie jest to wyjątkowa instrukcja, jest synonimem instrukcji xchg ax, ax.

Instrukcja hlt, zatrzymuje procesor aż do resetu, przerwań nie maskowalnych lub innych przerwań (zakładając, że przerwania są odblokowane) przychodzących .generalnie, nie powinniśmy używać tej instrukcji na IBM PC chyba, że rzeczywiście wiemy co robimy. Instrukcja ta nie jest odpowiednikiem instrukcji halt w x86.Nie używajmy jej do zatrzymywania naszych programów!

80x86 dostarcza innego przedrostka instrukcji ,lock, który podobnie jak instrukcja rep, wpływa na następujące instrukcje. jednakże. instrukcja ta ma trochę znaczeń na większości systemów PC .Jej celem jest koordynowanie systemów, które mają wiele CPU. Ponieważ systemy z wieloma CPU stają się dostępne ,ten przedrostek może w końcu okazać się cenny. Nie będziemy jednak zbytnio się tu na nim koncentrować.

Pentium dostarcza dwóch dodatkowych instrukcji interesujących programistów w trybie rzeczywistym DOS. Instrukcje te to cpuid i rdtsc .Jeśli ładujemy eax zerem i wykonujemy instrukcję cpuid, Pentium (i późniejsze procesory) zwróci wartość maksymalną cpuid przeznaczoną jako parametr w eax. Dla Pentium, tą wartością jest jeden. Jeśli ładujemy rejestr eax jedynką i wykonujemy instrukcję cpuid, Pentium zwróci informację identyfikującą CPU w eax.

Druga instrukcja Pentium interesująca to instrukcja rdtsc. Pentium utrzymuje 64 bitowy licznik, który liczy cykle zegarowe startując przy resecie. Instrukcja rdtsc kopiuje bieżącą wartość licznika do pary rejestrów edx:eax. Możemy użyć tej instrukcji do dokładnego odmierzania czasu sekwencji kodu.

Poza instrukcjami przedstawionym do tej pory,80286 i późniejsze procesory dostarczają zbioru instrukcji trybu chronionego. Tekst ten nie będzie zajmował się nimi ponieważ są one użyteczne dla tych którzy piszą systemy operacyjne. Nie będziemy musieli nawet używać tych instrukcji w naszych aplikacjach kiedy uruchomimy w trybie chronionym systemu operacyjnego jak Windows, UNIX lub OS/2.Instrukcje te są zarezerwowane dal tych, którzy chcą pisać takie systemy operacyjne i sterowniki do nich.

6.14 PODSUMOWANIE

Rodzina procesorów 80x86 dostarcza bogatego zbioru instrukcji CISC (komputer z pełną liczbą rozkazów).Członkowie rodziny procesorów 80x86 są generalnie zgodne z przyszłymi wersjami, w znaczeniu kolejnych procesorów wykonujących wszystkie instrukcje z poprzednich chipów. Programy pisane dla 80x86 generalnie pracują na wszystkich członkach rodziny, programy używające nowych instrukcji na 80286 będą działały na 80286 i późniejszych procesorach, ale nie na 8086.Podobnie programy które wykorzystują nowe instrukcje na 80386 będą działały na 80386 i późniejszych procesorach ale nie na wcześniejszych. I tak dalej.

Procesory opisane w tym rozdziale zawierają 8086/8088, 80286,80386,80486 i Pentium(80586).Intel również stworzył 80186,ale ten procesor nigdy nie był używany intensywnie w komputerach osobistych.

Zbiór instrukcji 80x86 jest bardzo łatwo podzielić na osiem kategorii

* Instrukcje przenoszenia danych
* Konwersja
* Instrukcje arytmetyczne
* Instrukcje logiczne, przesunięcia, obrotu i bitowe
* Instrukcje I/O
* Instrukcje łańcuchowe
* Instrukcje sterowania strumieniem danych w programie
* Instrukcje różne

Wiele instrukcji wpływa na kilka bitów flag w rejestrze flag 80x86.Pewne instrukcje mogą testować te flagi ponieważ są one wartościami boolowskimi. Flagi również oznaczają znaki po porównaniu takie jak równość, mniejszy niż i większy niż .Aby naumieć się o tych flagach i jak testujemy je w programie, zajrzyj

- Rejestr Stanu Procesora (Flagi)
- Instrukcje Warunkowe
- Instrukcje skoków warunkowych

Jest kilka instrukcji które przesyłają dane między rejestrami a pamięcią. Instrukcje te są jedynymi, których programista asemblerowy wzywa bardzo często.80x86 dostarcza wiele takich instrukcji które pomagają nam pisać szybsze i wydajniejsze programy. O szczegółach czytaj:

- Instrukcje przenoszenia danych
- Instrukcja MOV
- Instrukcja Xchg
- Instrukcje LDS,LES<LFS<LGS i LSS
- Instrukcja LEA
- Instrukcje PUSH i POP
- Instrukcje LAHF i SAHF

80x86 dostarcza kilku instrukcji do konwersji danych z jednej formy do innej. Jest kilka specjalnych instrukcji dla powielania znaku, powielania znaku, tablicy translacji i konwersji big/little endian.

- Instrukcje Movzx ,movsx, cbw, cwd, cwde i cdq
- Instrukcja bswap
- Instrukcja xlat

Instrukcje arytmetyczne 80x86 dostarczają wszystkich powszechnych działań: dodawania, mnożenia, odejmowania, dzielenia, negacji, porównania, zwiększania, zmniejszania i kilku instrukcji pomocnych przy arytmetyce BCD:AAA,AAD,AAM,AAS,DAA i DAS

- Instrukcje arytmetyczne
- Instrukcje dodawania: ADD,ADC,INC,XAAD,AAA i DAA
- Instrukcje ADD i ADC
- Instrukcja INC
- Instrukcja XADD
- Instrukcje odejmowania: SUB,SBB,DEC,AAS i DAS
- Instrukcja CMP
- Instrukcje CMPXCHG i CMPXCHG8B
- Instrukcja NEG
- Instrukcje mnożenia: MUL,IMUL i AAM
- Instrukcje dzielenia: DIV, IDIV i AAD

80x86 dostarcza również bogatego zbioru operacji logicznych, przesunięcia, obrotu i bitowych. Instrukcje te manipulują bitami w swoich operandach pozwalając na logiczne AND,OR,XOR i NOT wartości, obroty i przesunięcia bitów wewnątrz operandu, testowanie i ustawianie /zerowanie /odwracanie bitów w operandzie, ustawianie operandu na zero lub jeden w zależności od stanu rejestru flag

- Instrukcje Logiczne, Przesunięcia, Obrotu i Bitowe
- Instrukcje logiczne: AND,OR,XOR i NOT
- Instrukcje Obrotu: RCL,RCR,ROL i ROR
- Operacje bitowe
- Instrukcje warunkowe

W zbiorze instrukcji 80x86 jest para instrukcji I/O,IN i OUT .Są one właściwie specjalnymi formami instrukcji MOV która działa w przestrzeni adresowej I/O 80x86 zamiast w przestrzeni adresowej pamięci.

Typowym zastosowaniem tych instrukcji jest uzyskiwanie dostępu do rejestrów sprzętowych urządzeń peryferyjnych.

- Instrukcje I/O

Rodzina 80x86 dostarcza dużego repertuaru instrukcji które manipulują łańcuchami danych. Instrukcje te to movs, lods, stos, scas, cmps ,ins, outs, rep, repz, repe, repnz i repne.

- Instrukcje łańcuchowe

Instrukcje sterowania przesyłaniem danych w 80x86 pozwalają nam tworzyć pętle, podprogramy ,sekwencje warunków i wiele innych testów.

- Instrukcje sterowania przebiegiem programu
- Skoki Bezwarunkowe
- Instrukcje CALL i RET
- Instrukcje INT,INTO,BOUND i IRET
- Instrukcje skoków warunkowych
- Instrukcje JCXZ/JECXZ
- Instrukcja LOOP
- Instrukcja LOOPE/LOOPZ
- Instrukcja LOOPNE?LOOPNZ

Na końcu rozdział ten omawia kilka różnorodnych instrukcji. Instrukcje te bezpośrednio manipulują flagami w rejestrze flag, dostarczają specjalnej obsługi procesora lub wykonują operacje trybu chronionego. Ten rozdział tylko wspomina o instrukcjach trybu chronionego. Ponieważ zwykle nie będziemy ich używać w programach (nie –O/S) użytkowych, nie musimy o nich wiedzieć

- Instrukcje Różnorodne

6.15 PYTANIA

1. Dostarcz przykładu który pokazuje że wymagane jest n+11 bitów do przechowania sumy dwóch n-bitowych wartości
2. ADC i SBB mogą być wykorzystane do zachowania się jak ADD I SUB przez włożenie kilku innych instrukcji między ADC i SBB. Jaka instrukcja musi być zastosowana przed ADC aby zachowywała się jak ADD a jaka przed SBB aby zachowywała się jak SUB?
3. Przypuśćmy że możemy manipulować pozycjami danych na szczycie stosu używając PUSH i POP. Wyjaśnij jak możemy zmodyfikować adres powrotu na stosie żeby instrukcja RET spowodowała w 80x86 powrót dwóch bajtów poza oryginalny adres powrotu.
4. Dostarcz czterech różnych sposobów na dodanie dwóch wartości w rejestrze BX. Żaden sposób nie powinien wymagać więcej niż dwóch instrukcji(podpowiedź, jest co najmniej sześć sposobów zrobienia tego)
5. Załóżmy ,że adres docelowy dla następujących skoków warunkowych jest poza zakresem krótkiego rozgałęzienia. Zmodyfikuj każdą z tych instrukcji aby wykonywała właściwe działania
   a) JS Label1 b)JE Label2 c)JZ Label3 d)JC Label4 e)JBE There f)JG Label5
6. Wyjaśnij różnice pomiędzy flagą przeniesienia a flagą przepełnienia
7. Kiedy używamy instrukcji CBW i CWD do powielenia wartości znaku?
8. Jaka jest różnica pomiędzy instrukcjami „MOV reg, dana bezpośrednia” a „LEA reg, adres”
9. Co robi instrukcja INT nm odkładając na stos to czego instrukcja CALL FAR nie robi?
10. Jakie jest typowe zastosowanie instrukcji JCXZ
11. Wyjaśnij działanie instrukcji LOOP,LOOPE/LOOPZ i LOOPNE/LOOPNZ
12. Na jaki rejestr (inny niż rejestr flag) wpływają instrukcje MUL,IMUL,DU|IV i IDIV?
13. Wypisz trzy różnice pomiędzy DEC AX i SUB AX,1
14. Która z instrukcji przesunięcia, obrotu i logiczna nie wpływa na flagę zera?
15. Dlaczego instrukcja SAR zawsze zeruje flagę przepełnienia
16. Na 80386 instrukcja IMUL jest prawie całkowicie ortogonalna. Prawie. Daj kilka przykładów form dozwolonych dla instrukcji ADD dla których nie ma porównywalnych instrukcji IMUL
17. Dlaczego Intel uogólnił instrukcję IDIV a nie zrobił tego z instrukcją IMUL?
18. Jakiej instrukcji będziemy musieli użyć aby odczytać ośmio bitową wartość spod adresu I/O 378h?Proszę podać określoną instrukcję do zrobienia tego.
19. Której flagi(flag0 używa 80x86 do sprawdzenia bez znakowego przepełnienia arytmetycznego?

20. Jaka flaga (-i) pozwalają nam sprawdzić przepełnienie ze znakiem?
21. Jakiej flagi (flag) używa 80x86 do testowania następujących warunków bez znakowych? Ile musi być ustawionych flag aby warunek był prawdziwy?
    a) równy b) nie równy c) mniejszy niż d)mniejszy niż lub równy e)większy niż f) większy niż lub równy
22. Powtórz powyższe pytanie dla porównania ze znakiem
23. Wyjaśnij działanie instrukcji CALL i RET 80x86.Opisz krok po kroku każdy wariant tych instrukcji
24. Następująca sekwencja wymienia wartości pomiędzy dwoma zmiennymi pamięci I i J:
    xchg ax ,i
    xchg ax, j
    xchg ax, i
    Na 80486,instrukcje „MOV reg, mem" i „MOV mem, reg" zabierają tylko jeden cykl podczas gdy „Xchg reg, mem" zabiera trzy cykle. Pokaż szybszą sekwencję dla '486 niż powyższa.
25. Na 80386 instrukcja „MOV reg ,mem" wymaga czterech cykli ,"MOV mem ,reg" wymaga dwóch cykli a „Xchg reg, mem" wymaga pięciu cykli. Pokaż szybszą sekwencję problemu wymiany pamięci w pytaniu 24 dla 80386
26. Na 80486 instrukcje „MOV acc, mem" i „MOV reg, mem" wszystkie zabierają tylko jeden cykl do wykonania. Zakładając ,że wszystko inne jest dobre, dlaczego nie chcemy użyć „MOV acc, mem" zamiast „MOV reg, mem" dla załadowania wartości do AL./AX?EAX?
27. Jakie instrukcje wykonują ładowanie 32 bitów na procesorach przed-80386?
28. Jak możemy użyć instrukcji PUSH i POP dla zachowania rejestru AX pomiędzy dwoma punktami naszego kodu?
29. Jeśli bezpośrednio na wejściu do podprogramu wykonamy instrukcję pop ax, jaką wartość będziemy mieli w rejestrze AX?
30. Jaka jest główna zastosowanie dla instrukcji SAHF?
31. Jaka jest różnica między CWD i CWDE?
32. Instrukcja BSWAP skonwertuje 32 bitową wartość big endian na 32 bitową wartość little endian .Jakiej instrukcji możemy użyć do konwersji 16 bitowego big endian na 16 bitowy little endian**?**
33. Jaka instrukcja może być użyta do konwersji wartości 32 bitowego little endian na 32 bitowa wartość big endian?
34. Wyjaśnij jak możemy zastosować instrukcję XLAT do konwersji znaku alfabetycznego w rejestrze AL. z małej do dużej litery (zakładając ,że jest tam mała litera) i pozostawić wszystkie inne wartość w AL. niezmienione.
35. Jak instrukcja jest bardzo podobna do CMP?
36. Jak instrukcja jest bardzo podobna do TEST?
37. Co robi instrukcja NEG?
38. Pod jakimi warunkami zawiodą instrukcje DIV i IDIV?
39. Jaka jest różnica między RCL i ROL?
40. Napisz krótki segment kodu, używając instrukcji LOOP, który wywołuje „podprogram „CallMe" 25 razy.
41. Na 80486 i Pentium instrukcja LOOP nie jest tak szybka jak instrukcje dyskretne, które wykonują te same operacje .Przepisz powyższy kod do stworzenia szybciej wykonującego się programu na chipach 80486 i Pentium.
42. Jak możemy określić „przeciwny skok" dla skoku warunkowego. Dlaczego ten algorytm jest bardziej pożądany?
43. Podaj przykład instrukcji BOUND. Wyjaśnij co twój przykład będzie robił.
44. Jak jest różnica między instrukcjami IRET i RET (daleki)?
45. Instrukcja BT (Test Bitu) kopiuje określony bit do flagi przeniesienia. Jeśli określony bit to jedynka, ustawia ona flagę przeniesienia, jeśli ten bit to zero. zeruje flagę przeniesienia. Przypuśćmy ,że chcemy wyzerować flagę przeniesienia jeśli bit był zero, w przeciwnym razie ustawiony. Jak instrukcja może się wykonać po osiągnięciu tego przez BT?
46. Możemy zasymulować instrukcję dalekiego powrotu używając zmiennej podwójnego słowa i dwóch instrukcji 80x86.jak jest sekwencja tych dwóch instrukcji do osiągnięcia tego?