ROZDZIAŁ DZIEWIĄTY: **OPERACJE ARYTMETYCZNE I LOGICZNE**

Dużo więcej jest potrzebne do posługiwania się asemblerem niż znajomość mnogich operacji. Musimy nauczyć się jak je stosować i co one robią, Wiele instrukcji jest użytecznych dla operacji, które mają mało pracy z ich matematycznymi lub oczywistymi funkcjami .Rozdział ten omawia jak skonwertować wyrażenia z języka wysokiego poziomu do asemblera, Omawia również zaawansowane operacje arytmetyczne i logiczne wliczając w to wielokrotnej precyzji i sztuczki ,którymi możemy zabawić się z różnymi instrukcjami.

9.0 WSTĘP

Rozdział ten omawia sześć głównych tematów: konwertowanie wyrażeń arytmetycznych HLL'a na język asemblera, wyrażenia logiczne, arytmetyczne i logiczne operacje o podwyższonej precyzji, działanie na operandach o różnych rozmiarach, styl maszynowy i arytmetyczny i operacje maskowania. Podobnie jak poprzednie rozdziały, rozdział ten zawiera obszerny materiał, którego musimy się nauczyć natychmiast jeśli jesteśmy początkującymi programistami asemblerowymi. Poniższe części, które mają przedrostek „•" są niezbędne. te części z „⊗" omawiają zaawansowane tematy, które możemy odłożyć na jakiś czas.

- • Wyrażenia arytmetyczne
- • Proste przypisania
- • Proste wyrażenia
- • Wyrażenia złożone
- • Operatory przemienności
- • Wyrażenia logiczne
- • Operacje wielokrotnej precyzji
- • Operacje dodawania o wielokrotnej precyzji
- • Operacje odejmowania o wielokrotnej precyzji
- • Porównania o podwyższonej precyzji
- ⊗ Mnożenie o podwyższonej precyzji
- ⊗ Dzielenie o podwyższonej precyzji
- ⊗ Negacja o rozszerzonej precyzji
- • AND, OR, XOR i NOT o rozszerzonej precyzji
- ⊗ Operacje przesunięcia i obrotu o podwyższonej precyzji
- ⊗ Działanie na operandach o różnych rozmiarach
- • Mnożenie bez MUL i IMUL
- ⊗ Dzielenie bez DIV i IDIV
- ⊗ Zastosowanie AND do obliczania reszt
- ⊗ Licznik Modulo – n z AND
- ⊗ Testowanie dla 0FFFF...FFFh
- • Operacje testowe
- ⊗ Znaki testujące z instrukcją XOR
- ⊗ Operacje maskowania
- ⊗ Maskowanie z instrukcją AND

- ⊗ Maskowanie z instrukcją OR
- ⊗ Pakowanie i rozpakowywanie typów danych
- ⊗ Tablice połączeń

Żaden z tych materiałów nie jest szczególnie trudny do zrozumienia. Jednak, jest tu wiele nowych tematów a zabranie się za nie po kilka na raz pozwoli nam lepiej wchłonąć materiał. Te tematy z przedrostkiem „•" są jedynymi, stosowanymi przez nas często; Stąd jest dobrym pomysłem zacząć studiować je jako pierwsze.

9.1 WYRAŻENIA ARYTMETYCZNE

Prawdopodobnie największym szokiem dla początkujących odkrywających asembler po raz pierwszy jest brak dobrze znanych wyrażeń arytmetycznych. Wyrażenia arytmetyczne, w większości języków wysokiego poziomu ,wygląda podobnie do tego algebraicznego równania:

```
X :=  Y*Z;
```

W asemblerze, potrzebujemy kilku instrukcji do wykonania tego samego zadania np.

```
mov     ax, y
imul    z
mov     x, ax
```

Oczywiście wersja dla HLLi jest dużo łatwiejsza do napisania, czytania i zrozumienia. Punkt ten, bardziej niż inne, jest odpowiedzialny za odstraszanie ludzi od asemblera.

Chociaż jest wiele zawiłości, konwersja wyrażeń arytmetycznych do języka asemblera nie jest wcale trudne. Przez zaatakowanie problemu krok po kroku, chociaż może rozwiązalibyśmy problem ręcznie, możemy łatwo podzielić każde arytmetyczne wyrażenie na odpowiednią sekwencję instrukcji asemblerowych. Ucząc się jak skonwertować takie wyrażenia na asembler w trzech krokach ,odkryjemy, że jest to trochę trudniejsze zadanie.

9.1.1 PROSTE PRZYPISANIA

Najłatwiejszymi wyrażeniami do konwersji na język asemblera są proste przypisania. Proste przypisania kopiują pojedynczą wartość do zmiennej i przybierają jedną z dwóch form:

```
zmienna  :=  stała
```

lub

```
zmienna  :=  zmienna
```

Jeśli zmienna pojawia się w bieżącym segmencie danych (np. DSEG), konwersja pierwszej postaci na język asemblera jest łatwe, po prostu używamy instrukcji asemblera:

```
mov     zmienna , stała
```

Ta instrukcja move bezpośrednio kopiuje stałą do zmiennej.

Drugie powyższe przypisanie jest nieco bardziej skomplikowane ponieważ 80x86 nie dostarcza instrukcji mov pamięć do pamięci. Dlatego też, aby skopiować jedną zmienną pamięci do innej, musimy przesunąć dane przez rejestr. Jeśli spojrzymy na kodowanie dla instrukcji mov w dodatku, zauważymy ,że instrukcje mov ax, pamięć i mov pamięć, ax są krótsze niż przesuwanie wymagającego innych rejestrów. Na przykład,

```
var1  := var2
```

staje się

```
mov     ax, var2
mov     var1, ax
```

Oczywiście, jeśli zastosujemy rejestry ax do czegoś innego, wystarczy jeden z innych rejestrów. Mimo to musimy zastosować rejestr do przeniesienia jednej komórki pamięci do innej.

Omówienie to oczywiście zakłada, że obie zmienne są w pamięci. Jeśli to możliwe, powinniśmy spróbować stosować rejestr do przechowywania wartości zmiennej.

9.1.2 PROSTE WYRAŻENIA

Wyższym stopniem złożoności od prostego przypisania jest proste wyrażenie. Proste wyrażenie przybiera formę:

$$var := term_1 \; op \; term_2;$$

Var jest zmienną,$term_1$ i $term_2$ są zmiennymi lub stałymi a op jest jakimś arytmetycznym operatorem (dodawania odejmowania, mnożenia itp.)

Większość wyrażeń przyjmuje taką formę. Nie powinno być to dla nas niespodzianka, że architektura 80x86 została zoptymalizowana właśnie dla takiego typu wyrażeń.

Typowa konwersja dla tego typu wyrażenia przyjmuje postać:

```
mov     ax, term1
```

```
        op      ax, term2
        mov     var, ax
```

Op jest mnemonikiem, który odpowiada wyszczególnionej operacji (np. „+" = add, „-„ = sub, itp.)

Jest kilka niekonsekwencji, których musimy być świadomi. Po pierwsze, instrukcje {i}mul 80x86 nie pozwalają na operandy bezpośrednie na procesorach wcześniejszych niż 80286.Dalej,żaden procesor nie pozwala na bezpośredni operand z {i}div. Dlatego też, jeśli operacją jest mnożenie lub dzielenie a jeden z warunków jest wartością stałą, będziemy musieli załadować tą stałą do rejestru lub komórki pamięci a potem pomnożyć lub podzielić ax przez tą wartość. Oczywiście, kiedy zajmujemy się mnożenia i dzielenia na 8086/8088,musimy stosować rejestry ax i dz. Nie możemy zastosować przypadkowego rejestru jak możemy uczynić to z inna operacją .Również, nie zapomnijmy instrukcji rozszerzenia znaku jeśli wykonujemy operację dzielenia i dzielimy jedną 16/32 bitową liczbę przez inna. W końcu, nie zapomnijmy, że pewne instrukcje mogą powodować przepełnienie .Możemy chcieć sprawdzić warunek przepełnienia (lub niedomiaru) po operacji arytmetycznej.

Przykłady prostych wyrażeń:

```
X :=  Y + Z;
        mov     ax, Y
        add     ax, Z
        mov     x,  ax
X  :=  Y – Z;
        mov     ax, y
        sub     ax, z
        mov     x, ax
X  :=  Y * Z;   {bez znaku}
        mov     ax, y
        mul     z               ;stosujemy IMUL dla arytmetyki znakowej
        mov     x,  ax          ;nie zapomnijmy tym zmieść dx

X  := Y  div  Z;  {dzielenie bez znaku}
        mov     ax, y
        mov     dx, 0           ;rozszerzenie zerem AX do DX
        div     z
        mov     x,  ax
X  :=  Y  div  Z  {div ze znakiem}
        mov     ax,  y
        cwd                     ; rozszerzenie znakiem AX do DX
        idiv    z
        mov     x, ax
X  :=  Y  mod  Z {reszta bez znaku}
        mov     ax, y
        mov     dx, 0           ;rozszerzenie zerem AX do DX
        div     z
        mov     x, dx           ;reszta jest w DX
X  :=  Y  mod Z  { reszta ze znakiem}
        mov     ax, y
        cwd                     ;rozszerzenie znakiem AX do DX
        idiv    z
        mov     x, dx           ;reszta jest w DX
```

Ponieważ jest możliwe wystąpienie błędu arytmetycznego, powinniśmy ogólnie testować wynik każdego wyrażenia na błąd przed lub po ukończonej operacji .Na przykład, bez znakowe dodawanie, odejmowanie i mnożenie ustawiają flagę przeniesienia jeśli wystąpi przepełnienie. Możemy zastosować instrukcje bezpośrednio po odpowiedniej sekwencji instrukcji dla testu na przepełnienie. Podobnie, możemy użyć instrukcji jo lub jno po tej sekwencji do testowania dla przepełnienia przy arytmetyce ze znakiem .Następne dwa przykłady demonstrują jak zrobić to dla instrukcji add:

```
X  :=  Y + Z; {bez znaku}
        mov     ax, y
        add     ax,  z
        mov     x,  ax
        jc      uOverflow

X  :=  Y + Z; {ze znakiem}
```

```
mov     ax, y
add     ax, z
mov     x, ax
jo      sOverflow
```

pewne operacje jednoargumentowe również kwalifikują się jako proste wyrażenia. Dobrym przykładem operacji jednoargumentowej jest negacja .W językach wysokiego poziomu negacja przyjmuje jedną z dwóch możliwych form:

var := - var or var_1 := - var_2

Zauważmy, że var := - stała jest w rzeczywistości prostym przypisaniem, nie prostym wyrażeniem. Możemy wyszczególnić stałą ujemną jako operand instrukcji mov:

```
mov     var ,  -14
```

Przy operowaniu pierwszą formą operacji negacji stosujemy pojedynczą instrukcją asemblerową:

```
neg     var
```

Jeśli są stosowane dwie różne zmienne, wtedy stosujemy jak następuje:

```
mov     ax, var2
neg     ax
mov     var2, ax
```

Przepełnienie wystąpi wtedy gdy spróbujemy zanegować największą wartość (-128 dla wartości ośmio bitowej, -32768 dla szesnasto bitowej .itd.).W tym przypadku 80x86 ustawia flagę przepełnienia ,więc możemy testować dla przepełnienia arytmetycznego stosując instrukcje jo lub jno We wszystkich innych przypadkach 80x86 zeruje flagę przepełnienia. Flaga przeniesienia nie ma znaczenia po wykonaniu instrukcji neg ponieważ neg (oczywiście) nie stosuje operandu bez znakowego.

9.1.3 WYRAŻENIA ZŁOŻONE

Wyrażenie złożone jest każdym wyrażeniem arytmetycznym zawierającym więcej niż dwa warunki i jeden operator. Takie wyrażeniami są powszechnie znajdowane w programach pisanych w językach wysokiego poziomu. Wyrażenia złożone może zawierać nawiasy z operatorami pierwszeństwa ,wywołania funkcji, dostęp do tablic itp. Konwersja jakiegoś złożonego wyrażenia na język asemblera jest dosyć proste, inne wymagają wysiłku. Ta sekcja naszkicuje zasady jakich używamy przy konwersji takich wyrażeń.

Złożona funkcja która jest łatwa do konwersji na asembler jest funkcją, która zawiera trzy warunki i dwa operatory ,na przykład:

W := W- Y – Z;

Konwersja tej instrukcji na język asemblera wymaga dwóch instrukcji sub. Jednakże, nawet w wyrażeniu tak prostym jak to, konwersja nie jest banalna. Faktycznie są dwa sposoby konwersji powyższej instrukcji na asembler:

```
mov     ax, w
sub     ax, y
sub     ax, z
mov     w, ax
```

i

```
mov     ax, y
sub     ax, z
sub     w, ax
```

Druga konwersja, ponieważ jest krótsza ,wygląda lepiej. Jednakże, tworzy ona niepoprawny wynik (zakładając Pascalową składnię dla oryginalnej instrukcji).Problemem jest prawo łączności. Druga z powyższych sekwencji oblicza W:=W-(Y-Z),która nie jest tym samym co W:=(W-Y)-Z .Jak umieścimy nawiasy wokół podwyrażeń możemy wpływać na wynik. Zauważmy, że jeśli jesteśmy zainteresowani krótszą forma, możemy zastosować poniższą sekwencję:

```
mov     ax, y
add     ax, z
sub     w, ax
```

Oblicza to W:=W-(Y+Z).Jest to odpowiednik W:=(W-Y)-Z.

Inną kwestią jest pierwszeństwo. Rozważmy wyrażenie Pascalowskie:

X := W*Y+Z;

Znowu mamy dwa sposoby w jaki możemy ocenić to wyrażenie:

X := (W*Y)+Z;

lub

X := W*(Y+Z)

Prawdopodobnie teraz myślisz, że ten tekst jest szalony. Każdy wie, że poprawnym sposobem oceny tych wyrażeń jest druga forma przedstawiona w tym drugim przykładzie. Jednakże ,mylisz się, jeśli myślisz w ten

sposób .Język programowania APL ,na przykład, ocenia wyrażenia wyłącznie od prawej do lewej i nie daje pierwszeństwa jednemu operatorowi przed innym.

Większość języków wysokiego poziomu używa stałego zbioru zasad pierwszeństwa opisujących porządek oceniania w wyrażeniach wymagających dwóch lub więcej różnych operatorów. Większość języków programowania oblicza mnożenie i dzielenie przed dodawaniem i odejmowaniem. Te które wspierają podnoszenie do potęgi (np. FORTRAN i BASIC) zazwyczaj obliczają je przed mnożeniem i dzieleniem. Zasady te są intuicyjne ponieważ prawie każdy uczył się o nich w szkole. Rozważmy wyrażenie:

$X \ op_1 \ Y \ op_2 \ Z$

Jeśli op_1 posiada pierwszeństwo przed op2,wtedy wyliczamy to $(X \ op_1 \ Y) \ op_2 \ Z$ w przeciwnym razie jeśli op_2 ma pierwszeństwo przed op_1 wtedy wyliczamy to jako $X \ op_1 \ (Y \ op_2 \ Z)$.W zależności od tego czy są wymagane operatory i operandy te dwa obliczenia mogą tworzyć różne wyniki.

Kiedy konwertujemy wyrażenie tej postaci do języka asemblera, musimy najpierw obliczyć podwyrażenie z najwyższym priorytetem. Poniższy przykład demonstruje tą technikę:

```
; W  := X+Y *Z;
                mov     bx, x
                mov     ax, y           ;musimy najpierw obliczyć Y*Z ponieważ „*” ma najwyższy
                mul     z               ;priorytet.
                add     bx, ax          ;teraz  dodajemy iloczyn i wartość Xa
                mov     w, bx           ;zachowujemy wynik
```

Ponieważ dodawanie jest operacją przemienną ,możemy zoptymalizować powyższy kod tworząc:

```
; W  :=  X+Y*Z;
                mov     ax, y           ;musimy obliczyć najpierw Y*Z ponieważ „*” ma najwyższy
                mul     z               ;priorytet.
                add     ax, x           ;teraz dodajemy iloczyn i wartość Xa
                mov     w, ax           ;zachowujemy wynik
```

Jeśli dwa operatory pojawiające się wewnątrz wyrażenia mają taki sam priorytet, wtedy określamy porządek wyliczania używając zasady łączności .Większość operatorów jest lewo łącznych w znaczeniu, że są wyliczane od lewej do prawej Dodawanie, odejmowanie, mnożenie i dzielenie wszystkie są lewo łączne. Prawo łączne operatory są wyliczane od prawej do lewej. Operator potęgowania w FORTRANie i BASICu jest dobrym przykładem operatora prawo łącznego:

2^2^3 jest równy 2^(2^3) nie (2^2)^3

Zasady pierwszeństwa i łączności określają porządek wyliczania. Pośrednio te zasady mówią nam gdzie umieścić nawiasy w wyrażeniu dla określenia pierwszeństwa i łączności. Jednak, ostatecznie nasz kod asemblerowy musi ukończyć pewne działania przed innymi aby poprawnie wyliczyć wartość danego wyrażenia. Poniższy przykład demonstruje ta zasadę:

```
; W  := X-Y-Z
                mov     ax, x           ;wszystkie operatory są takie same, więc musimy obliczać
                sub     ax, y           ;od lewej do prawej ponieważ wszystkie maja taki
                sub     ax,  z          ;sam priorytet
                mov     w,  ax
;W  := X+Y*Z
                mov     ax,y            ,najpierw musimy obliczyć Y*Z ponieważ mnożenie ma
                imul    z               ;wyższy priorytet niż dodawanie
                add     ax, x
                mov     w,  ax
;W  := X/Y –Z
                mov     ax, x           ;tu musimy najpierw obliczyć dzielenie  ponieważ ma wyższy
                cwd                     ;priorytet
                idiv    y
                sub     ax, z
                mov     w,  ax
;W  := X*Y*Z
                mov     ax, y           ;dodawanie i mnożenie są przemienne, dlatego porządek obliczania
                imul    z               ;nie ma znaczenia
                imul    x
                mov     w,  ax
```

Jest jeden wyjątek od zasady łączności. Jeśli wyrażenie wymaga mnożenia i dzielenia, zawsze lepiej najpierw wykonać mnożenie .Na przykład, dane jest wyrażenie w postaci:

W := X/Y *Z

Lepiej jest obliczyć X*Z a potem podzielić wynik przez Y zamiast dzielić X przez Y i pomnożyć iloraz przez Z. Są dwa powody, że takie podejście jest lepsze. Po pierwsze, pamiętajmy, że instrukcja imul zawsze tworzy wynik 32 bitowy (zakładając 16 bitowy operand)Przez wykonanie najpierw mnożenia automatycznie powielamy znak iloczynu do rejestru dx aby nie musieć powielać znaku ax wcześniejszego dzielenia. To zapewnia wykonanie instrukcji cwd. Drugi powód wykonania najpierw mnożenia ,to zwiększenie precyzji obliczeń .Pamiętamy, że dzielenia (całkowite) często tworzy wynik niedokładny. Na przykład, jeśli obliczymy 5/2 otrzymamy wartość dwa a nie 2.5.Obliczenie (5/2)*3 da nam sześć. Jednakże, jeśli obliczamy (5*3)/2 dostaniemy wartość siedem, która jest trochę zbliżona do rzeczywistej wartości (7.5).Dlatego też, jeśli napotkamy wyrażenie w postaci:

W := X/Y*Z

Zazwyczaj możemy ją skonwertować do kodu asemblerowego:

```
mov     ax, x
imul    z
idiv    z
mov     w, ax
```

Oczywiście, jeśli algorytm jakim kodujemy zależy od efektu zaokrąglenia operacji dzielenia, nie możemy zastosować tej sztuczki do poprawienia algorytmu. Morał z tej historii: zawsze upewniajmy się ,czy dokładnie zrozumieliśmy dane wyrażenia, które chcemy skonwertować do języka asemblera. Oczywista, że jeśli semantyka dyktuje, że najpierw musimy wykonać dzielenie, zróbmy to.

Rozważmy poniższa instrukcję pascalowską:

W := X – Y *Z;

Jest ona podobna do poprzedniego przykładu z wyjątkiem tego, że stosujemy odejmowanie zamiast dodawania. Ponieważ odejmowanie nie jest przemienne, nie możemy obliczyć Y*Z a potem odjąć X od tego wyniku. Trochę to nam skomplikuje nieco konwersję .Zamiast prostej sekwencji mnożenia i dodawania ,musimy załadować X do rejestru, pomnożyć Y i Z pozostawiając iloczyn w innym rejestrze a potem odjąć ten iloczyn od X np.

```
mov     bx, x
mov     ax, y
imul    z
sub     bx, ax
mov     w, bx
```

Jest to trywialny przykład, który demonstruje potrzebę stosowania zmiennych tymczasowych w wyrażeniu. Kod stosujący rejestr bx tymczasowo przechowuje kopię X dopóki obliczany jest iloczyn Y i Z. Jeśli nasze wyrażenia stają się coraz bardziej złożone, rośnie potrzeba na tymczasowość rośnie. Rozważmy poniższą instrukcję Pascalowską:

W := (A+B)*(Y+Z);

Stosując zwykłe zasady oceniania, obliczamy najpierw podwyrażenia wewnątrz nawiasów (tj. dwa podwyrażenia z najwyższymi priorytetami) i rezerwujemy miejsce w pamięci dla ich wartości. Kiedy obliczymy wartości dla obu podwyrażeń, możemy obliczyć ich sumę. Jedyny sposób zajęcia się złożonym wyrażeniem takim jak to jest zredukowanie go do sekwencji prostych wyrażeń, których wyniki przetrzymuje się w zmiennych tymczasowych. Na przykład, możemy skonwertować powyższe pojedyncze wyrażenie do następującej sekwencji:

$Temp_1$:= A+B;;
$Temp_2$:= Y+Z;
W := $Temp_1$ * $Temp_2$;

Ponieważ konwertowanie prostych wyrażeń do języka asemblera jest całkiem łatwe, teraz dopasowujemy obliczenia pierwszego, złożonego wyrażenia w asemblerze. Kod :

```
mov     ax, a
add     ax, b
mov     temp1, ax
mov     ax, y
add     ax, z
mov     temp2, ax
mov     ax, temp1
imul    temp2
mov     w, ax
```

Oczywiście, kod ten jest rażąco niewydajny i wymaga zadeklarowania pary zmiennych tymczasowych w segmencie danych. Jednakże jest łatwy sposób zoptymalizowania tego kodu przez przetrzymanie zmiennych tymczasowych, tak długo jak to możliwe w rejestrach 80x86.Poprzez zastosowanie rejestrów 80x86, przechowujących wyniki tymczasowe te kod staje się taki;

```
mov     ax, a
add     ax, b
mov     bx, y
add     bx, z
imul    bx
mov     w, ax
```

Jeszcze inny przykład:

X := (Y+Z)*(A-B) / 10

Będzie to skonwertowane po ustaleniu czterech prostych wyrażeń:

Temp1 := (Y+Z)
Temp2 := (A-B)
Temp1 := Temp1 * Temp2
X := Temp1 / 10

Możemy skonwertować te cztery proste wyrażenia na instrukcje języka asemblera:

```
mov     ax, y           ;oblicza AX  := Y+Z
add     ax, z
mov     bx, a           ;oblicza BX  := A-B
sub     bx, b
mul     bx              ;oblicza  AX := AX*BX, również powiela znak
mov     bx, 10          ;AX do DX dla iciv
idiv    bx              ;oblicza AX  := AX / 10
mov     x, ax           ;przechowuje wynik w X
```

Najważniejsza rzecz jaka jest do zapamiętania to taka ,że wartości tymczasowe, jeśli to możliwe, powinny być trzymane w rejestrach. Pamiętamy, że dostęp do rejestrów 80x86 jest dużo bardziej wydajny niż dostęp do komórek pamięci .Stosujemy komórki pamięci do przechowywania tymczasówek, tylko jeśli korzystamy już z rejestrów.

Ostatecznie, konwertowanie złożonych wyrażeń na język asemblera, trochę różni się od rozwiązywania wyrażeń ręcznie. Zamiast właściwego obliczania wyniku w każdej fazie obliczeń, po prostu piszemy kod asemblerowy, który oblicza wyniki Ponieważ nauczyliśmy się obliczać tylko jedno działania na raz, to znaczy, że ręczne obliczenia pracują nad „prostym wyrażeniem”, które istnieje w wyrażeniu złożonym. Oczywiście konwertowanie tych prostych wyrażeń na asembler jest banalnie proste. Dlatego też, każdy kto może rozwiązać złożone wyrażenie ręcznie, może skonwertować go na język asemblera korzystając z zasad dla prostych wyrażeń.

9.1.4 OPERATORY PRZEMIENNOŚCI

Jeśli „@” przedstawia jakiś operator, ten operator jest przemienny, jeśli następujący związek zawsze jest prawdziwy:

(A @ B) = (B @ A)

Jak widzieliśmy w poprzedniej sekcji, operatory przemienności są przyjemne, ponieważ porządek ich operandów jest nieistotny a to pozwala nam przestawiać obliczenia, często czyniąc to obliczenie łatwiejszym lub bardziej wydajnym. Często przestawianie obliczeń pozwala nam na zastosowanie mniej zmiennych tymczasowych. Kiedykolwiek napotkamy operator przemienności w wyrażeniu, powinniśmy zawsze sprawdzić czy istnieje lepsza sekwencja, którą można zastosować do poprawy szybkości naszego kodu. Poniższa tabela wylicza operatory przemienności i nie przemienne jakie zwykle znajdujemy w językach wysokiego poziomu:

Pascal	C/C++	Description
+	+	Addition
*	*	Multiplication
AND	&& or &	Logical or bitwise AND
OR	|| or |	Logical or bitwise OR
XOR	^	(Logical or) Bitwise exclusive-OR
=	==	Equality
<>	!=	Inequality

Tablica 46 : Popularne binarne operatory przemienności

Pascal	C/C++	Description
-	-	Subtraction
/ or DIV	/	Division
MOD	%	Modulo or remainder
<	<	Less than
<=	<=	Less than or equal
>	>	Greater than
>=	>=	Greater than or equal

Tablica 47 : Popularne nie przemienne operatory binarne

9.2 WYRAŻENIA LOGICZNE (BOOLOWSKIE)

Rozważmy następujące wyrażenie z programu pascalowskiego:

B:= ((X=Y) and (A <= C)) or ((Z-A) <> 5);

B jest zmienną boolowską a pozostałe zmienne są całkowite.

Jak przedstawimy zmienne boolowskie w języku asemblera? Chociaż, przyjmują tylko jeden bit dla przedstawienia wartości boolowskiej, większość programistów asemblerowych przeznacza cały bajt lub słowo na ten cel. Z bajtem, jest 256 możliwych wartości, jakie możemy zastosować dla przedstawienia dwóch wartości prawda i fałsz. Wiec która z dwóch wartości (lub które dwa zbiory wartości) stosujemy do przedstawienia tych wartości boolowskich? Z powodu architektury maszyny, jest dużo łatwiej testować dla warunków takich jak zero lub nie-zero i dodatnich lub ujemnych zamiast dla jednej lub dwóch szczególnych wartości boolowskich. Większość programistów (a nawet języki programowani takie jak C) wybiera zero do przedstawiania fałszu i coś innego dla przedstawiania prawdy .niektórzy ludzie wolą przedstawiać prawdę i fałsz jako jeden i zero (odpowiednio) i nie pozwalają na inne wartości. Inni wybierają 0FFFFh dla prawdy i 0 dla fałszu. Możemy również zastosować wartości dodatnie dla prawdy i ujemne dla fałszu. wszystkie te mechanizmy mają zalety i wady.

Używanie tylko zera i jedynki dla przedstawiania fałszu i prawdy oferuje jedną bardzo dużą zaletę: instrukcje logiczne 80x86 (and, or, xor i , w mniejszym stopniu, not) działają na tych wartościach dokładnie tak jak od nich oczekujemy. to znaczy, jeśli mamy dwie zmienne boolowskie A i B, wtedy poniższe instrukcje wykonują podstawowe operacje logiczne na tych dwóch zmiennych:

```
mov     ax, A
and     ax, B
mov     C, ax           ;C := A and B

mov     ax, A
or      ax, B
mov     C, ax           ;C := A or B

mov     ax, A
xor     ax, B
mov     C, ax           ;C  := A xor B

mov     ax, A
not     ax
and     ax, 1
mov     B, ax           ;B := not A

mov     ax, A           ;inny sposób zrobienia B := NOT A
xor     ax, 1
mov     B, ax           ;B  := not A
```

Zauważ, jak wskazano powyżej, że instrukcja not nie dokładnie oblicza logiczną negację. Not zera na poziomie bitowym to 0FFh a not jeden na poziomie bitowym to 0FEh.Żaden wynik nie jest zerem lub jedynką. Jednakże,

przez dodanie jeden do wyniku, uzyskamy wynik poprawny. Zauważmy, że możemy uczynić operację not bardziej wydajną używając instrukcji xor ax,1 ponieważ wpływa ona tylko na najmniej znaczący bit. Okazuje się, że stosując zero dla fałszu i jakiejś innej wartości dla prawdy mamy dużo subtelnych zalet .Ściśle, test dla prawdy lub fałszu jest często ukryty w wykonywaniu każdej logicznej instrukcji. Jednak ten mechanizm cierpi na bardzo dużą niedogodność :nie możemy używać instrukcji and ,or, xor i not 80x86 do implementacji działań boolowskich tej samej nazwy. Rozpatrzmy dwie wartości 55h i 0Aah.Obie są niezerowe, więc obie przedstawiają wartość prawdy .Jednak, jeśli logicznie zandujemy 55h i 0Aah razem, stosując instrukcję and 80x86,wynikiem będzie zero.(Prawda i prawda) powinny tworzyć prawdę ,nie fałsz. System, który stosuje wartości niezerowych do przedstawiania prawdy i zera dla przedstawiania fałszu jest arytmetycznym systemem logicznym. System, który stosuje dwie odrębne wartości, takie jak zero i jeden do przedstawiania fałszu i prawdy jest nazywany boolowskim systemem logicznym, lub po prostu, systemem boolowskim. Możemy zastosować inny system, równie dogodny .Rozważmy znów wyrażenie boolowskie:

```
B  := ((X=Y) and (A<=D)) or ((Z-A)  <>5);
```

Proste wyrażenia wynikłe z tego wyrażenia mogą być następujące:

```
Temp2  := X=Y
Temp  := A <= D
Temp  := Temp  and  Temp2
Temp2  := Z-A
Temp2  := Temp2 <> 5
B := Temp  or Temp2
```

Kod asemblerowy dla tego wyrażenia może być taki:

```
        mov     ax, x           ;patrzy czy X + Y i ładuje zero lub jeden do AX
        cmp     ax, y           ;oznaczający wynik tego porównania.
        jnz     L1
        mov     al., 1          ; X = Y
        jmp     L2
L1:     mov     al., o          ;X <>Y
L2:
        mov     bx, A           ;patrzy czy A <= D i ładuje zero lub jeden do BX
        cmp     bx, D           ;wynik tego porównania
        jle     ST1
        mov     bl, 0
        jmp     L3
ST1:    mov     bl, 1
L3:
        and     bl, al.         ;Temp := Temp  and Temp2

        mov     ax, Z           ;czy (Z-A) <>5
        sub     ax, A           ;Temp2 := Z-A
        cmp     ax, 5           ;Temp2  := Temp2 <>5
        jnz     ST2
        mov     al., 0
        jmp     short L4
ST2:    mov     al., 1
L4:
        or      al., bl         ;Temp := Temp  or Temp2
        mov     B, al.          ;B := Temp
```

Jak możemy zobaczyć jest to dosyć nieporęczna sekwencja instrukcji. Jedną niewielką optymalizacja jaką możemy zastosować, jest założenie ,że wynik będzie prawdą lub fałszem i zainicjowanie odpowiedniego wyniku boolowskiego przed czasem:

```
        mov     bl, 0           ;Zakładamy X <> Y
        mov     ax, x
        cmp     ax, Y
        jne     L1
        mov     bl, 1           ;X jest równe Y więc jest to prawda
L1:
        mov     bh, 0           ;zakładamy ,że nie (A <=D)
        mov     ax, A
        cmp     ax, D
```

```
                jnle    L2
                mov     bh, 1                   ;A <= D  więc jest to prawda
L2:
                and     bl, bh                  ;obliczamy wynik logicznego AND

                mov     bh,0                    ;zakładamy ,że  (Z-A) = 5
                mov     ax, Z
                sub     ax, A
                cmp     ax, 5
                je      L3:
                mov     bh, 1                   ;(Z-A) <> 5
L3:
                or      bl, bh                  ; wynik logicznego OR
                mov     B, bl                   ;zachowujemy wynik boolowski
```

Oczywiście, jeśli mamy 80386 lub późniejszy procesor ,możemy zastosować instrukcje setcc do uproszczenia tego trochę:

```
                mov     ax, x
                cmp     ax, y
                sete    al.                     ;Temp2 := X=Y

                mov     bx, A
                cmp     bx, D
                setle   bl                      ;Temp := A <= D
                and     bl, al.                 ;Temp  := Temp  and Temp2
                mov     ax, Z
                sub     ax, A                   ;Temp2  := Z-A
                cmp     ax, 5                   ;Temp2  := Temp2  <> 5
                setne   al.
                or      bl, al.                 ;Temp:=Temp  or Temp2
                mov     B, bl                   ;B := Temp
```

Ta sekwencja kodu jest oczywiście dużo lepsza niż poprzednie, ale wykonuje się tylko na pocesorach80386 i późniejszych.

Innym sposobem zajęcia się wyrażeniami boolowskimi jest przedstawianie wartości boolowskich poprzez stany wewnątrz naszego kodu. Podstawową ideą jest zapomnieć o utrzymaniu zmiennych boolowskich przez całe wykonywanie sekwencji kodu i zastosowanie lokalizacji wewnątrz kodu do określenia wyniku boolowskiego. Rozważmy następująca implementację powyższego wyrażenia .Po pierwsze, poprzestawiamy wyrażenie:

```
                B := ((Z-A) <> 5) or ((X=Y) and (A <=D));
```

Jest to zupełnie poprawne ponieważ operator or jest przemienny teraz rozważmy poniższą implementację:

```
                mov     B, 1                    ;zakładamy ,że wynik jest prawdą
                mov     ax, Z                   ;sprawdzamy czy (Z-A) <> 5
                sub     ax, A                   ;jeśli ten warunek jest prawdziwy, wynik jest zawsze
                cmp     ax, 5                   ;prawdą i nie musimy sprawdzać reszty
                jne     Done

                mov     ax, X                   ;jeśli X <> Y wynik jest fałszem
                cmp     ax, Y                   ;bez względu co zawiera A i D
                jne     SetBtoFalse

                mov     ax, A                   ;czy A <= D
                cmp     ax, D
                jle     Done                    ;jeśli tak, to wyjście
SetBtoFalse:    mov     B, 0
Done:
```

Zauważmy, że ta sekcja kodu jest dużo krótsza iż pierwsza wersja powyżej (i działa na wszystkich procesorach).Poprzednia robiła wszystko obliczeniowo. Ta wersja stosuje logikę działań programu do poprawy kodu. Zaczyna od założenia prawdziwości wyniku i ustawia zmienną B na prawda. potem sprawdza czy (Z-A) <>5.Jeśli jest to prawda, kod rozgałęzia się do tablicy done ponieważ B jest prawdą bez względu na to co się wydarzy. jeśli program nie dochodzi do instrukcji mov ax, X, wiemy, że wynik poprzedniego porównania jest

fałszem. Nie musimy zachowywać tego wyniku w zmiennej tymczasowej ponieważ pośrednio znamy wynik poprzez fakt, że wykonujemy instrukcję mov ax, X. Podobnie ,druga grupa instrukcji sprawdza czy X jest równe Y .Jeśli nie, już wiemy, że wynik jest fałszem, więc ten kod skacze do etykiety SetBtoFalse .Jeśli program zaczyna wykonywanie trzeciego zbioru instrukcji ,wiemy ,że pierwszy wynik był fałszem a drugi wynik był prawdą: położenie kodu to gwarantuje. Dlatego nie musimy utrzymywać tymczasowej zmiennej boolowskiej, która śledzi stan tego obliczania.

Rozważmy inny przykład;

B := ((A=E) or (F <>D) and ((A<>B) or (F=D)

Obliczeniowo, to wyrażenie daje znaczną ilość kodu. Jednak ,poprzez użycie sterowania strumieniem danych możemy zredukować go jak następuje:

```
                mov     b, 0            ;zakładamy wynik fałsz
                mov     ax, a           ;czy A = E
                cmp     ax, e
                je      test2           ;jeśli tak, pierwsze podwyrażenie jest prawdą

                mov     ax, f           ;jeśli nie sprawdź drugie podwyrażenie
                cmp     ax, d           ;czy F <> D
                je      Done            ;jeśli tak przechodzimy do drugiego testu
Test2:          mov     ax, a           ;czy A <> B?
                cmp     ax, b
                jne     SetBtol         ;jeśli tak, zrobiono

                mov     ax, f           ;jeśli nie, zobacz czy F = D
                cmp     ax, d
                jne     Done
SetBtol:        mov     b, 1
Done:
```

Jest jedna różnica pomiędzy stosowaniem sterowaniem przebiegiem programu a logiką obliczeniowa: kiedy stosujemy metody sterowania przebiegiem programu, możemy pominąć większość instrukcji, które implementuje formuła boolowska. Jest to znane jako obliczenie częściowe. kiedy stosujemy model obliczeniowy, nawet z instrukcją setcc, kończymy wykonywanie większości instrukcji. Zapamiętajmy, że nie jest to koniecznie wada. Na procesorach potokowych, można dużo szybciej wykonać kilka dodatkowych instrukcji zamiast opróżniać potok i kolejkę rozkazów. może musimy poeksperymentować z naszym kodem dla określenia najlepszego rozwiązania.

Kiedy pracujemy z wyrażeniami boolowskimi, nie zapomnijmy, że można zoptymalizować nasz kod poprzez upraszczanie tych wyrażeń boolowskich (zobacz „Upraszczanie Funkcji Boolowskich").możemy zastosować transformację algebraiczną (zwłaszcza teorie DeMorgana) i metodę mapowania do pomocy przy redukcji złożonych wyrażeń.

9.3 OPERACJE WIELOKROTNEJ PRECYZJI

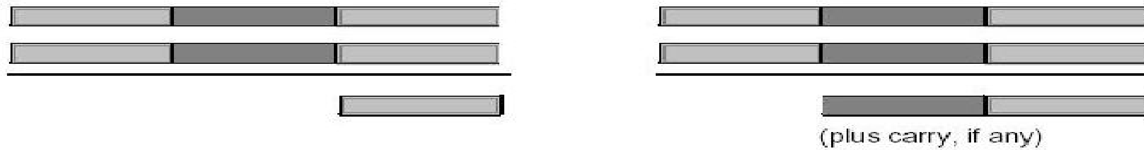
Jedną wielką zaletą języka asemblera nad HLL'ami jest to, że asembler nie ogranicza wielkości liczb całkowitych. Na przykład, język C definiuje maksymalnie trzy różne wielkości całkowite: short int, int i long int .Na PC są często 16 lub 32 bitowe liczby całkowite .Chociaż instrukcje maszynowe 80x86 ograniczają nas do działania na ośmio- szesnasto lub trzydziesto dwu bitowych liczbach całkowitych z pojedynczą instrukcją, możemy zawsze zastosować więcej niż jedną instrukcję do działania na liczbach całkowitych o każdej wielkości jaką sobie życzymy. Jeśli chcemy wartość całkowitą 256 bitową, żaden problem. Poniższa sekcja opisuje jak rozszerzyć różne arytmetyczne i logiczne operacje z 16 lub 32 bitów do tak wielu bitów jaka nas zadowala.

9.3.1 OPERACJE DODAWANIA O WIELKOKTROTNEJ PRECYZJI

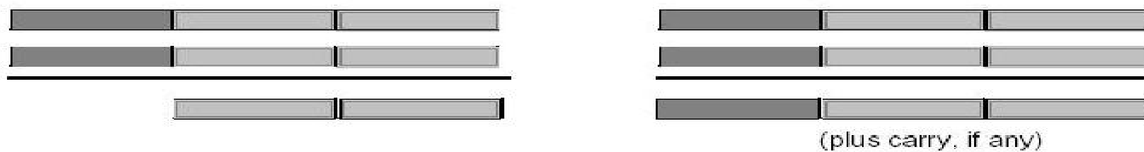
Instrukcja add 80x86 dodaje dwa 8, 16 lub 32 bitowe liczby. Po wykonaniu instrukcji dodawania, flaga przeniesienia jest ustawiana jeśli wystąpi przepełnienie z najbardziej znaczącego bitu sumy.

Step 1: Add the least significant words together:

Step 2: Add the middle words together:



Step 3: Add the most significant words together:



Rysunek 9.1 dodawanie wielokrotnej precyzji (48 bitowe)

Możemy użyć tych informacji dla operacji dodawania wielokrotnej precyzji. Rozważmy sposób ręcznego wykonania operacji dodawania wielocyfrowego (wielokrotnej precyzji):

Krok1: Dodajemy razem najmniej znaczące cyfry :

```
 289                          289
+456        produces         +456
----                         ----
                               5 with carry 1.
```

Krok2: Dodajemy następne znaczące cyfry plus przeniesienie:

```
 1 (previous carry)
289                                 289
+456              produces         +456
----                               ----
  5                                  45 with carry 1.
```

Krok 3: Dodajemy najbardziej znaczące cyfry plus przeniesienie:

```
                                    1 (previous carry)
 289                                289
+456              produces         +456
----                               ----
  45                                745
```

80x86 operuje rozszerzoną precyzją arytmetyki w identycznej formie, z wyjątkiem tego, że zamiast dodawania lic cyfr, dodaje bajt lub słowo. Rozważmy operację dodawania trój słowa (48 bitów) z rysunku 8.1. Instrukcja add dodaje najmniej znaczące słowa razem. Instrukcja adc (dodawanie z przeniesieniem) dodaje inne pary słów razem. Instrukcja adc dodaje dwa operandy plus flaga przeniesienia razem tworząc wartość słowa i (możliwe) przeniesienie.

Na przykład przypuśćmy, że mamy dwie trzydziesto dwu bitowe wartości, które życzymy sobie dodać razem, zdefiniowane jak następuje:

```
X	dword	?
Y	dword	?
```

Przypuśćmy też, że chcemy przechować sumę w trzeciej zmiennej, Z, która jest podobnie zdefiniowana dyrektywą dword. poniższy kod 80x86 realizuje to zadanie:

```
        mov     ax, word ptr X
        add     ax, word ptr Y
        mov     word ptr Z, ax
        mov     ax, word ptr X+2
        adc     ax, word ptr Y+2
        mov     word ptr Z+2, ax
```

Pamiętamy, że te zmienne są deklarowane dyrektywą dword. Dlatego też asembler nie zaakceptuje instrukcji w postaci mov ax, X ponieważ instrukcja ta próbuje załadować 32 bitową wartość do 16 bitowego rejestru. Dlatego też kod ten stosuje operator koercji word ptr do sprowadzenia symboli X,Y i Z do szesnastu bitów. Pierwsze trzy instrukcje dodają razem najmniej znaczące słowa X i Y i przechowują wynik w najmniej znaczącym słowie Z. Ostatnie trzy instrukcje dodają najbardziej znaczące słowa X i Y razem z przeniesieniem z mniej znaczącego słowa, i przechowują wynik w bardziej znaczącym słowie Z. Pamiętajmy, że wyrażenia adresowe w postaci „X+2" uzyskuje dostęp do bardziej znaczącego słowa 32 bitowej jednostki. To z powodu faktu, że przestrzeń adresowa 80x86 jest adresowana bajtami i zabiera dwa kolejne bajty na słowo.

Oczywiście, jeśli mamy 80386 lub późniejsze procesory, nie musimy przechodzić przez wszystkie dodawania dwóch 32 bitowych wartości razem ,ponieważ 80386 bezpośrednio wspiera 32 bitowe operacje .jednakże, jeśli chcielibyśmy dodać razem dwie 64 bitowe wartości całkowite na 80386,będziemy musieli zastosować tę technikę.

Możemy rozszerzyć to na każdą liczbę bitów poprzez zastosowanie instrukcji adc dla dodawania w wyższym porządku słów w wartości. Na przykład, dodanie razem dwóch 128 bitową wartość ,możemy użyć kod, który wygląda podobnie jak poniższy::

```
BigVal1         dword   0,0,0,0                 ;cztery podwójne słowa
BigVal2         dword   0,0,0,0
BigVal3         dword   0,0,0,0
                -
                -
                -
                mov     eax, BigVal1            ;nie potrzebujemy operatora dword ptr ponieważ są to
                add     eax,  Bigval2           ; zmienne dword
                mov     BigVal3, eax

                mov     eax,BigVal+4
                adc     eax,BigVal2+4
                mov     BigVal3+4, eax

                mov     eax,BigVal1+8
                adc     eax,BigVal2+8
                mov     BigVal3+8, eax

                mov     eax,BigVal1+12
                adc     eax,bigVal2+12
                mov     BigVal3+12, eax
```

9.3.2 OPERACJE ODEJMOWANIE WIELOKROTNEJ PRECYZJI

Podobnie jak dodawanie,80x86 wykonuje odejmowanie wielobajtowe, w ten sam sposób ręcznie, z wyjątkiem kiedy odejmuje całe bajty, słowa lub podwójne słowa na raz zamiast cyfr dziesiętnych. Mechanizm jest podobny do tego dla operacji dodawania.. Stosujemy instrukcje sub na mniej znaczącym bajcie /słowie /podwójnym słowie a instrukcję sbb na wartościach bardziej znaczących cyfr .Poniższy przykład demonstruje 32 bitowe odejmowanie stosując 16 bitowe rejestry na 8086:

```
var1            dword   ?
var2            dword   ?
diff            dword   ?

                mov     ax, word ptr var1
                sub     ax, word ptr var2
                mov     word ptr diff, ax
                mov     ax, word ptr var1+2
                sbb     ax, word ptr var2+2
                mov     word ptr diff+2, ax
```

Poniższy przykład demonstruje 128 bitowe odejmowanie stosując zbiór 32 bitowych rejestrów 80386:

```
BigVal1        dword   0,0,0,0
BigVal2        dword   0,0,0,0
BigVal3        dword   0,0,0,0
               -
               -
               -
               mov     eax,BigVal1          ,nie potrzebujemy operatora dword ptr ponieważ są to
               sub     eax, BigVal2         ;zmienne podwójnego słowa
               mov     BigVal3, eax

               mov     eax,BigVal1+4        ;odejmujemy wartości od  mniej do bardziej znaczącej
               sbb     eax, BigVal2+4       ;jednostki stosując instrukcje SUB i SBB
               mov     BigVal3+4, eax

               mov     eax, BigVal1+8
               sbb     eax, BigVal2+8
               mov     BigVal3+8,eax

               mov     eax, BigVal1+12
               sbb     eax,BigVal2+12
               mov     BigVal3+12, eax
```

9.3.3 PORÓWNANIA O ROZSZERZONEJ PRECYZJI

Niestety nie ma instrukcji „porównania z pożyczką", która mogła by być zastosowana dla porównania o rozszerzonej precyzji. Ponieważ instrukcje cmp i sub wykonują takie same operacje, przynajmniej jeśli chodzi o flagi „prawdopodobnie domyślamy się, że można użyć instrukcji sbb do syntezy porównań o rozszerzonej precyzji; Jednakże jest to częściowa prawda .Jest lepszy sposób.

Rozważmy dwie wartości bez znaku 2157h i 1293h.Mniej znaczące bajty tych dwóch wartości nie wpływają na wynik porównania. Po prostu porównujemy 21h z 12h,które mówią nam ,że pierwsza wartość jest większa niż druga. Faktycznie, jedyny raz kiedy musimy spojrzeć na oba bajty tych wartości jest wtedy czy bardziej znaczące bajty są równe. We wszystkich innych przypadkach porównania bardziej znaczących bajtów mówią nam wszystko co musimy wiedzieć o wartościach. Oczywiście ,jest to prawda dla każdej liczby bajtów, nie tylko dwóch. Poniższy kod porównuje dwie 64 bitowe liczby całkowite ze znakiem na 80386 i późniejszych procesorach:

```
;Jest to  przekazanie sterowania do lokacji „IsGreater" jeśli QwordValue > QwordValue2
;Przekazuje sterowanie do „IsLess" jeśli Qword Value < QwordValue2.nie dochodzi do wykonania
;tych instrukcji jeśli qwordValue = QwordValue2.Test dla nierówności zmienia operandy „IsGreater" i
;"IsLess" na „NotEquaL" w tym kodzie.
               mov     eax, dword ptr QwordValue+4        ;pobiera bardziej znaczący dword
               cmp     eax, dword ptr QwordValue2+4
               jg      IsGreater
               jl      IsLess
               mov     eax, dword ptr QwordValue
               cmp     eax, dword ptr QwordValue2
               jg      IsGreater
               jl      IsLess
```

Dla porównania wartości bez znakowych stosujemy instrukcje ja i jb w miejsce jg i jl

Możemy łatwo syntetyzować każde możliwe porównanie z powyższą sekwencją, poniższy przykład pokazuje jak to zrobić te przykłady robisz porównanie ze znakiem, zastępując ja ,jae, jb i jbe za jg, jge i jle (odpowiednio) dla porównania bez znakowego

```
QW1     qword   ?
QW2     qword   ?

dp      textequ  <dword ptr>

;testujemy 64 bity aby zobaczyć  czy QW! < QW2 (ze znakiem)
;przekazuje sterownie do etykiety  „IsLess' jeśli QW1 < QW2..Nie dochodzi do skutku następna instrukcja jeśli
;nie jest to prawdą
        mov     eax, dp QW1+4                ;pobiera bardziej znaczący dword
```

```
        cmp     eax, dp QW2+4
        jg      NotLess
        jl      IsLess
        mov     eax, dp QW1                 ;nie dochodzi do skutku jeśli bardziej znaczące dwordy
        cmp     eax, dp QW2                 ;są równe
        jl      IsLess
NotLess:

;testujemy 64 bity aby zobaczyć czy QW1 <= QW2 (ze znakiem)
        mov     eax, dp QW1+4               ;pobieramy bardziej znaczący dword
        cmp     eax, dp QW2+4
        jg      NotLessEq
        jl      IsLessEq
        mov     eax, dp QW1
        cmp     eax, dword ptr QW2
        jle     IsLessEq

NotLessEQ:

;testujemy 64 bity aby zobaczyć czy QW1 > QW2 (ze znakiem)
        mov     eax, dp QW1+4               ;pobranie bardziej znaczącego dworda
        cmp     eax, dp QW2+4
        jg      IsGtr
        jl      NotGtr
        mov     eax, dp QW1                 ;nie dochodzi do skutku jeśli bardziej znaczące dwordy
        cmp     eax, dp QW2                 ;są równe
        jg      IsGtr

NotGtr:

;testujemy 64 bity aby zobaczyć czy QW1 >= QW2 (ze znakiem)
        mov     eax, dp QW1+4                       ;pobranie bardziej znaczącego dworda
        cmp     eax, dp QW2+4
        jg      IsGtreq
        jl      NotGtrEq
        mov     eax, dp QW1
        cmp     eax, dword ptr QW2
        jge     IsGtrEq

NotGtrEq:

;testujemy 64 bity aby zobaczyć czy QW1 = QW2 (ze znakiem lub bez znaku).kod ten rozgałęzia się do etykiety
;"IsEqual" jeśli QW1 = QW2.Nie dochodzi do następnej instrukcji jeśli nie są one równe
        mov     eax, dp QW1+4
        cmp     eax, dp QW2+4
        jne     NotEqual
        mov     eax, dp QW1
        cmp     eax, dword ptr QW2
        je      IsEqual

NotEqual:

;testujemy 64 bity aby zobaczyć czy QW1 <> QW2 (ze znakiem lub bez znaku)Ten kod rozgałęzia się do
;etykiety „NotEqual" jeśli QW1 <> QW2.Nie dochodzi do skutku następna instrukcja jeśli są równe
        mov     eax, dp QW1 +4                      ;pobranie bardziej znaczącego dworda
        cmp     eax, dp QW2+4
        jne     NotEqual
        mov     eax, dp QW1
        cmp     eax, dword ptr QW2
        jne     NotEqual
```

9.3.4 MNOZENIE O ROZSZERZONEJ PRECYZJI

Chociaż mnożenie 16x16 lub 32x32 jest wystarczające, są chwile, kiedy chcemy pomnożyć razem. Większe wartości .Zastosujemy pojedynczy operand 80x86 instrukcji mul i imul dla mnożenia o rozszerzonej precyzji.

Bez zaskoczenia (zważywszy na to jak pracują adc i sbb), zastosujemy tą samą technikę dla wykonania mnożenia o rozszerzonej precyzji na 80x86,które stosujemy kiedy ręcznie mnożymy dwie wartości.

Rozważmy uproszczoną postać sposobu w jaki wykonujemy wielocyfrowe mnożenie ręcznie:

1) Mnożymy pierwsze dwie liczby:

```
123
 45
---
 15
```

2) Mnożymy 5*2:

```
123
 45
---
 15
10
```

3) Mnożymy 5*1

```
123
 45
---
 15
10
5
```

4)

```
4*3

123
 45
---
 15
10
5
12
```

5) Mnożymy 4*2

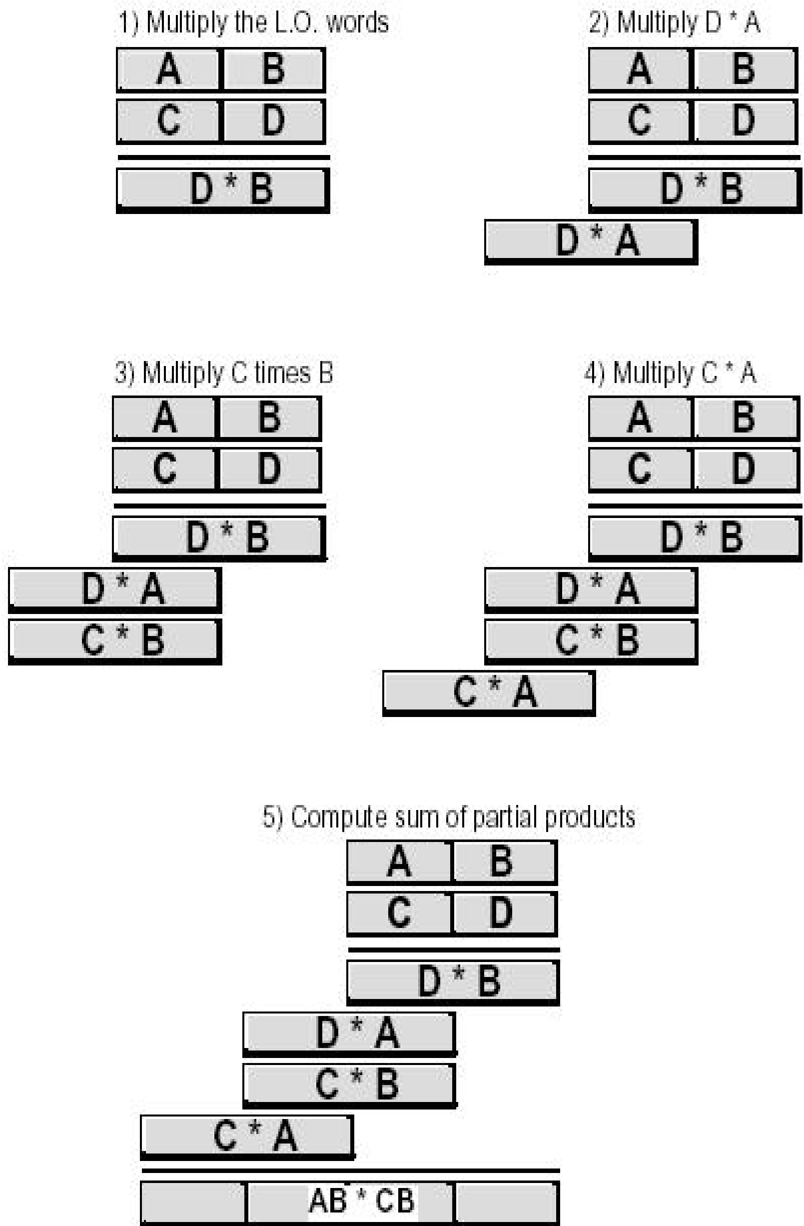
```
123
 45
---
 15
10
5
12
8
```

6) 4*1

```
123
 45
---
 15
10
5
12
8
4
```

7)dodajemy wszystkie części razem:

```
 123
  45
 ---
  15
 10
 5
 12
 8
4
------
5535
```



Rysunek 9.2 Mnożenie o zwielokrotnionej precyzji

80x86 robi emnożeni o wielokrotnej precyzji w ten sam sposób z wyjątkiem, tego ,że pracuje z bajtami, słowami i podwójnymi słowami zamiast cyframi. Rysunek 8.2 pokazuje jak to robi.

Prawdopodobnie najważniejszą rzeczą do zapamiętania kiedy wykonujemy mnożenie o rozszerzonej precyzji jest taka, że musimy również wykonać dodawanie o wielokrotnej precyzji w tym samym czasie. Dodanie wszystkich części iloczynu wymaga kilku dodawań .które stworzą wynik. poniższy listing demonstruje właściwy sposób mnożenia dwóch 32 bitowych wartości na szesnastobitowym procesorze:

Notka:Multiplier i Multiplicand są 32 bitowymi zmiennymi zadeklarowanymi w segmencie danych ,przez dyrektywę dword .Iloczyn jest 64 bitową zmienną zadeklarowaną w segmencie danych przez dyrektywę qword.

```
Multiply        proc    near
                push    ax
                push    dx
                push    cx
                push    bx

; Multiply the L.O. word of Multiplier times Multiplicand:

                mov     ax, word ptr Multiplier
                mov     bx, ax                          ;Save Multiplier val
                mul     word ptr Multiplicand           ;Multiply L.O. words
                mov     word ptr Product, ax            ;Save partial product
                mov     cx, dx                          ;Save H.O. word

                mov     ax, bx                          ;Get Multiplier in BX
                mul     word ptr Multiplicand+2         ;Multiply L.O. * H.O.
                add     ax, cx                          ;Add partial product
                adc     dx, 0                           ;Don't forget carry!
                mov     bx, ax                          ;Save partial product
                mov     cx, dx                          ; for now.

; Multiply the H.O. word of Multiplier times Multiplicand:

                mov     ax, word ptr Multiplier+2       ;Get H.O. Multiplier
                mul     word ptr Multiplicand           ;Times L.O. word
                add     ax, bx                          ;Add partial product
                mov     word ptr product+2, ax          ;Save partial product
                adc     cx, dx                          ;Add in carry/H.O.!

                mov     ax, word ptr Multiplier+2       ;Multiply the H.O.
                mul     word ptr Multiplicand+2         ; words together.
                add     ax, cx                          ;Add partial product
                adc     dx, 0                           ;Don't forget carry!
                mov     word ptr Product+4, ax          ;Save partial product
                mov     word ptr Product+6, dx

                pop     bx
                pop     cx
                pop     dx
                pop     ax
                ret
Multiply        endp
```

Jedną rzecz dotyczącą tego kodu musimy zapamiętać, pracuje tylko dla operandów bez znakowych.

9.3.5 DZIELENNIE O ROZSZERZONEJ PRECYZJI

Nie możemy zastosować ogólnej operacji dzielenia n-bitów /m.-bitów stosując instrukcji div i idiv. Taka operacja musi być wykonana przy zastosowaniu sekwencji instrukcji przesunięć i odejmowania. Tak operacja jest niezmiernie niechlujna. Mniej ogólna operacja, dzielenie n bitową wielkość przez 32 bitową (na 80386 lub późniejszych) lub 16 bitową wielkość jest łatwiejsza do zrobienia stosując instrukcję div .Poniższy kod demonstruje jak podzielić 64 bitową wielkość przez 16 bitowy dzielnik, tworząc 64 bitowy iloraz i 16 bitową resztę:

```
dseg        segment para public 'DATA'
divident    dword   0FFFFFFFFh, 12345678h
divisor     word    16
Qoutient    dword   0,0
Modulo      word    0
dseg        ends

cseg        segment para public 'CODE'
            assume  cs:cseg, ds.:dseg
;Dzielimy wielkość 64 bitową przez wielkość 16 bitową:

Divide64    proc    near
            mov     ax, word ptr dividend+6
            div     divisor
            mov     word ptr Quotient+6, ax
            mov     ax, word ptr dividend+4
            div     divisor
            mov     word ptr Quotient+4, ax
            mov     ax, word ptr dividend+2
            div     divisor
            mov     word ptr Quotient+2, ax
```

```
                mov     ax, word ptr dividend
                div     divisor
                mov     word ptr Quoyient, ax
                mov     Modulo, dx
                ret
Divide64        endp
cseg            ends
```

Kod ten może być rozszerzony do każdej liczby bitów przez proste dodanie dodatkowych instrukcji mov / div/ mov na początku sekwencji. Oczywiście, na 80386 i późniejszych procesorach możemy dzielić przez wartości 32 bitowe stosując edx i eax w powyższej sekwencji (z kilkoma innymi stosownymi regulacjami)

Jeśli musimy zastosować dzielnik większy niż 16 bitów (32 bity na 80386 i późniejszych) będziemy musieli zaimplementować dzielenie stosując strategię przesunięć i odejmowania. Niestety takie algorytmy są bardzo powolne. W tej sekcji rozwiniemy dwa algorytmy dzielenia, które działają na dowolnej liczbie bitów. Pierwszy jest wolny, ale łatwiejszy do zrozumienia, drugi jest trochę szybszy (ogólnie rzecz biorąc)

Podobnie jak dla mnożenia, najlepszy sposób zrozumienia jak komputer wykonuje dzielenie jest przestudiowanie jak nauczyć się wykonuje się długie dzielenie ręcznie .rozważmy działanie 3456 / 12 i krok po kroku wykonamy ręcznie tą operację:

```
12|3456     (1) 12 goes into 34 two times.
   24

    2
12|3456     (2) Subtract 24 from 35
   24           and drop down the
   105          105.

    2
12|3456     (3) 12 goes into 105
   24           eight times.
   105
    96

    28
12|3456     (4) Subtract 96 from 105
   24           and drop down the 96.
   105
    96
     96

    28
12|3456     (5) 12 goes into 96
   24           exactly eight times.
   105
    96
     96
     96

   288
12|3456     (6) Therefore, 12
   24           goes into 3456
   105          exactly 288 times.
    96
     96
     96
```

Algorytm ten jest w rzeczywistości łatwiejszy w systemie binarnym, ponieważ w każdym kroku nie musimy się domyślać ile razy 12 mieści się w reszcie oraz czy musimy mnożyć 12 przez domyślną liczbę odejmowań. Przy każdym kroku w algorytmie binarnym ,dzielnik zawiera resztę dokładnie zero lub jeden raz jako przykład rozpatrzmy dzielenie 27 (11011) przez trzy (11):

```
11|11011     11 goes into 11 one time.
   11
```

```
    1
11 |11011      Subtract out the 11 and bring down the zero.
    11
     00
```

```
    1
11 |11011      11 goes into 00 zero times.
    11
     00
     00
```

```
    10
11 |11011      Subtract out the zero and bring down the one.
    11
     00
     00
      01
```

```
    10
11 |11011      11 goes into 01 zero times.
    11
     00
     00
      01
      00
```

```
    100
11 |11011
    11
     00
     00        Subtract out the zero and bring down the one.
      01
      00
       11
```

```
    100
11 |11011
    11
     00
     00        11 goes into 11 one time.
      01
      00
       11
       11
```

```
    1001
11 |11011
    11
     00
     00        This produces the final result
      01       of 1001.
      00
       11
       11
       00
```

Jest to nowatorski sposób implementacji tego algorytmu binarnego dzielenia, który oblicza iloraz i resztę w tym samym czasie. Algorytm jest następujący:

```
Quotient := Dividend;
Remainder := 0;
for i:= 1 to NumberBits do

        Remainder:Quotient := Remainder:Quotient SHL 1;
        if Remainder >= Divisor then

                Remainder := Remainder - Divisor;
                Quotient := Quotient + 1;

        endif
endfor
```

NumberBits jest liczbą bitów w zmiennych Remainder, Quotient,Divisor i Dividend .Zauważmy, że instrukcja Quotient := Quotient +1 ustawia najmniej znaczący bit Quotient na jeden ponieważ ten poprzedni algorytm przesunął Quotient jeden bit w lewo .Kod 80x86 implementuje ten algorytm tak:

```
; Assume Dividend (and Quotient) is DX:AX, Divisor is in CX:BX,
; and Remainder is in SI:DI.
                mov     bp, 32          ;Count off 32 bits in BP
                sub     si, si          ;Set remainder to zero
                sub     di, di
BitLoop:        shl     ax, 1           ;See the section on shifts
                rcl     dx, 1           ; that describes how this
                rcl     di, 1           ; 64 bit SHL operation works
                rcl     si, 1
                cmp     si, cx          ;Compare H.O. words of Rem,
                ja      GoesInto        ; Divisor.
                jb      TryNext
                cmp     di, bx          ;Compare L.O. words.
                jb      TryNext
GoesInto:       sub     di, bx          ;Remainder := Remainder -
                sbb     si, cx          ;                 Divisor
                inc     ax              ;Set L.O. bit of AX
TryNext:        dec     bp              ;Repeat 32 times.
                jne     BitLoop
```

Ten kod wygląda na krótki i prostym ale, jest kilka problemów a nim .Po pierwsze, nie sprowadza dzielenia przez zero (tworzy wartość 0FFFFFFFFh jeśli próbujemy dzielić przez zero),działa tylko z wartościami bez znakowymi i jest bardzo wolny .Dzielenie przez zero jest bardzo proste, sprawdzamy czy dzielnik jest zerem podczas wcześniejszego wykonywania kodu i zwracamy właściwy kod błędu jeśli dzielnik jest zerem. Działanie z wartościami ze znakiem jest równie proste, co zobaczymy trochę później. Wydajność tego algorytmu pozostawia wiele do życzenia .Zakładając że jedno przejście przez pętlę zabiera 30 cykli zegarowych, ten algorytm wymagałby prawie 1000 cykli zegarowych! To jest porządny rozmiar, gorszy niż instrukcje DIV/IDIV na 80x86,które są wśród najwolniejszych instrukcji 80x86

Jest technika, którą możemy zastosować do zwiększenia wydajności tego dzielenia przy dużej ilości danych :sprawdzamy czy zmienna dzielnika rzeczywiście używa 32 bitów. Często nawet mimo, że dzielnik jest zmienną 32 bitową, jej wartość mieści się w 16 bitach.(np. bardziej znaczące słowo Divisora jest zerem)W tym specjalnym przypadku, który występuje często, możemy zastosować instrukcję DIV, która jest dużo szybsza.

9.3.6 OPERACJA NEG O ROZSZERZONEJ PRECYZJI.

Chociaż jest kilka sposobów zanegowania wartości o rozszerzonej precyzji, najkrótszym sposobem jest zastosowanie instrukcji neg lub sbb. technika ta korzysta z faktu, że neg odejmuje swój operand od zera. W szczególności, ustawia flagi w ten sam sposób jak instrukcja sub jeśli odjęlibyśmy wartość przeznaczenia od zera. Kod przyjmuje następującą postać:

```
neg     dx
neg     ax,
sbb     dx, 0
```

Instrukcja sbb zmniejsza dx jeśli wystąpi pożyczka z mniej znaczącego słowa operacji negacji (która zawsze wystąpi o ile ax nie jest zerem)

Rozszerzenie tej operacji do dodatkowego słowa lub podwójnego słowa jest łatwe; wszystko co musimy zrobić to zacząć od bardziej znaczącej komórki pamięci obiektu, który chcemy zanegować i działać w kierunku mniej znaczącego bajtu. Poniższy kod oblicza 128 bitową negację na procesorze 80386:

```
Value           dword   0,0,0,0         ;128 bit integer.
                .
                .
                .
                neg     Value+12        ;Neg H.O. dword
                neg     Value+8         ;Neg previous dword in memory.
                sbb     Value+12, 0     ;Adjust H.O. dword
                neg     Value+4         ;Neg the second dword in object.
                sbb     Value+8, 0      ;Adjust 3rd dword in object.
                sbb     Value+12, 0     ;Carry any borrow through H.O. word.
                neg     Value           ;Negate L.O. word.
                sbb     Value+4, 0      ;Adjust 2nd dword in object.
                sbb     Value+8, 0      ;Adjust 3rd dword in object.
                sbb     Value+12, 0     ;Carry any borrow through H.O. word.
```

Niestety, kod ten ma tendencje do stawania się rzeczywiście dużym i wolnym ponieważ musimy rozszerzyć przeniesienie przez wszystkie bardziej znaczące słowa po każdej operacji negacji. Prostszym sposobem zanegowania dużej wartości jest po prostu odjęcie tej wartości od zera:

```
Value           dword   0,0,0,0,0       ;160 bit integer.
                .
                .
                .
                mov     eax, 0
                sub     eax, Value
                mov     Value, eax
                mov     eax, 0
                sbb     eax, Value+4
                mov     Value+8, ax
                mov     eax, 0
                sbb     eax, Value+8
                mov     Value+8, ax
                mov     eax, 0
                sbb     eax, Value+12
                mov     Value+12, ax
                mov     eax, 0
                sbb     eax, Value+16
                mov     Value+16, ax
```

9.3.7 OPERACJA AND O ROZSZERZONEJ PRECYZJI

Wykonanie operacji and na n-słowach jest bardzo proste – po prostu andujemy odpowiadające sobie słowa pomiędzy dwoma operandami, zachowujemy wynik. Na przykład, wykonanie operacji and gdzie wszystkie trzy operandy są długości 32 bitów może wymagać takiego kodu:

```
mov     ax, word ptr source1
and     ax, word ptr source2
mov     word ptr dest, ax
mov     ax, word ptr source1+2
and     ax, word ptr source2+2
mov     word ptr dest+2, ax
```

Technika ta łatwo rozszerza się na każdą liczbę słów, wszystkie muszą zrobić logiczne andowanie odpowiadających bajtów, słów lub podwójnych słów w odpowiadających operandach.

9.3.8 OPERACJA OR O ROZSZERZONEJ PRECYZJI

Wielosłowna operacja logiczna or wykonuje się w ten sam sposób jak wielosłowna operacja and .Po prostu Orujemy odpowiadające sobie słowa w dwóch operandach. Na przykład, dla logicznego xorowania dwóch 48 bitowych wartości możemy zastosować taki kod:

```
mov     ax, word ptr operand1
or      ax, word ptr operand2
mov     word ptr operand3, ax
mov     ax, word ptr operand1+2
or      ax, word ptr operand2+2
mov     word ptr operand3+2, ax
mov     ax, word ptr operand1+4
or      ax, word ptr operand2+4
mov     word ptr operand3+4, ax
```

9.3.9 OPERACJA XOR O ROZSZERZONEJ PRECYZJI

Operacja xor o rozszerzonej precyzji wykonywana jest w sposób identyczny jak and /or, po prostu xorujemy odpowiadające sobie słowa w dwóch operandach uzyskując wynik o rozszerzonej precyzji. Poniższa sekwencja kodu działa na dwóch 64 bitowych operandach, oblicza ich exclusive-or i przechowuje wynik w 64 bitowej zmiennej Przykład ten stosuje 32 bitowe rejestry dostępne na 80386 i późniejszych

```
mov     eax, dword ptr operand1
xor     eax, dword ptr operand2
mov     dword ptr operand3, eax
mov     eax, dword ptr operand1+4
xor     eax, dword ptr operand2+4
mov     dword ptr operand3+4, eax
```

9.3.10 OPERACJA NOT O ROZSZERZONEJ PRECYZJI

Instrukcja not odwraca wszystkie bity w wyszczególnionym operandzie. Nie wpływa na żadną flagę (dlatego też, stosowanie skoku warunkowego po instrukcji not nie ma znaczenia)Not o rozszerzonej precyzji jest wykonywane przez proste wykonanie instrukcji not na wszystkich operandach. Na przykład, dla wykonania 32 bitowej operacji not na wartości w (dx:ax),wszystko co musimy zrobić to wykonać instrukcje:

```
Not     ax      lub     not     dx
Not     dx              not     ax
```

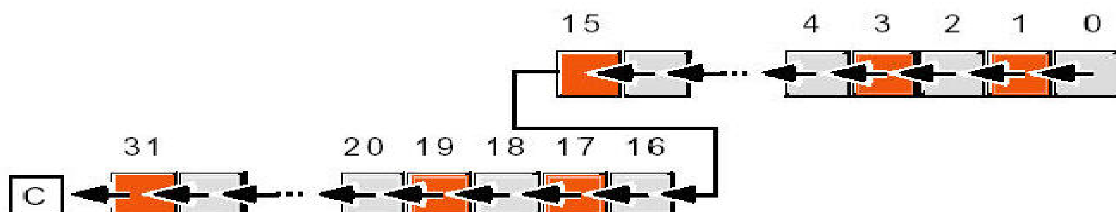
Zapamiętajmy, że jeśli wykonujemy instrukcję not dwa razy, kończymy z oryginalną wartością. Zauważmy również ,że xorownaie wartości wszystkich jedynek (0FFh,0FFFFh lub 0FF...FFh) wykonuje tą samą operację jak instrukcja not

---

9.3.11 OPERACJE PRZESUNIĘCIA O PODWYŻSZONEJ DOKŁADNOŚCI

Operacje przesunięcia o podwyższonej dokładności wymagają instrukcji przesunięcia i obrotu. Rozważmy co musi zdarzyć się dla implementacji 32 bitowego shl stosując 16 bitowe operacje

1) Zero musi być przesunięte do bitu zero
2) Bity od zera do 14 są przesuwane do następnego wyższego bitu
3) Bit 15 jest przesuwany do bitu 16
4) Bity 16 do 30 muszą być przesunięte do następnego wyższego bitu
5) Bit 31 jest przesuwany do flagi przeniesienia



Rysunek 9.3 Operacja 32 bitowego przesunięcia w lewo

Dwie instrukcje możemy zastosować do implementacji tego 32 bitowego przesunięcia to shl i rcl. Na przykład, dla przesunięcia 32 bitowej wielkości w (dx:ax) o jedną pozycję w lewo, zastosujemy takie instrukcje;

```
        shl     ax, 1
        rcl     dx, 1
```

Zauważmy, że możemy tylko przesunąć wartość o podwyższonej dokładności jeden bit na raz. Nie możemy przesunąć operandu o podwyższonej dokładności o kilka bitów stosując rejestr cl lub wartość bezpośrednią większą niż jeden ponieważ liczymy stosując tą technikę.

Dla zrozumienia jak ta sekwencja instrukcji pracuje rozważmy działanie tych instrukcji na osobnych podstawach. Instrukcja shl przesuwa zero do bitu zero 32 bitowego operandu a bit 15 do flagi przeniesienia Instrukcja rcl wtedy przesuwa flagę przeniesienia do bitu 16 a bit 31 do flagi przeniesienia. Wynik jest dokładnie tym czego oczekujemy.

Wykonanie operacji przesunięcia w lewo na operandzie większym niż 32 bity, to po prostu dodanie dodatkowej instrukcji rcl. Operacja przesunięcia w lewo o podwyższonej dokładności zawsze zaczyna się od najmniej znaczącego słowa a każda następna instrukcja rcl działa na następnym bardziej znaczącym słowie. Na przykład ,dla wykonania operacji 48 bitowego przesunięcia w lewo na komórce pamięci zastosujemy poniższe instrukcje:

```
        shl     word ptr Operand, 1
        rcl     word ptr Operand+2, 1
        rcl     word ptr Operand +4, 1
```

Jeśli musimy przesunąć nasze dane o dwa lub więcej bitów ,możemy albo powtórzyć powyższą sekwencję żądaną liczbę razy (dla stałej liczby przesunięć) albo możemy umieścić instrukcje w pętli do powtarzania jakąś liczbę razy, Na przykład, poniższy kod przesuwa wartość 48 bitową Operand w lewo o liczbę bitów wyszczególnioną w cx:

```
ShiftLoop:  shl     word ptr Operand, 1
            rcl     word ptr Operand+2, 1
            rcl     word ptr Operand+4, 1
            loop    ShiftLoop
```

Implementujemy shr i sar w podobny sposób z wyjątkiem tego, że musimy zacząć od bardziej znaczącego słowa operandu i pracować w kierunku mniej znaczącego słowa:

```
DblSAR:     sar     word ptr Operand+4, 1
            rcr     word ptr Operand+2, 1
            rcr     word ptr Operand, 1
DblSHR:     shr     word ptr Operand+4, 1
            rcr     word ptr Operand+2, 1
            rcr     word ptr Operand, 1
```

Jest jedna główna różnica między przesunięciami o podwyższonej dokładności opisanym tu a ich 8/16 bitowymi odpowiednikami – przesunięcia o podwyższonej dokładności ustawiają flagi inaczej niż operacja o pojedynczej dokładności. Na przykład, flaga zera jest ustawiana jeśli ostatnia instrukcja obrotu tworzy wynik zerowy, a nie jeśli cała operacja przesunięcia tworzy wynik zerowy. dla operacji przesunięcia w prawo, flagi przepełnienia i znaku nie są ustawiane poprawnie (są one ustawiane poprawnie dla przesunięcia w lewo).Dodatkowe testowanie będzie wymagane jeśli musimy przetestować jedną z tych flag po operacji przesunięcia o podwyższonej dokładności. Na szczęście, flaga przeniesienia jest flagą bardzo często testowaną po operacji przesunięcia a instrukcje przesunięcia o podwyższonej dokładności właściwie ustawiają te flagi.

Instrukcje shld i shrd pozwalają nam wydajniej implementować przesunięcia o zwielokrotnionej dokładności kilku bitów na 80386 i późniejszych procesorach. Rozważmy poniższą sekwencję kodu:

```
ShiftMe         dword   1234h, 5678h, 9012h
                 .
                 .
                 .
                mov     eax, ShiftMe+4
                shld    ShiftMe+8, eax, 6
                mov     eax, ShiftMe
                shld    ShiftMe+4, eax, 6
                shl     ShiftMe, 6
```

Pamiętajmy, że instrukcja shld przesuwa bity ze swego drugiego operandu do pierwszego operandu. Dlatego też, pierwsza powyższa instrukcja shld przesuwa bity od ShiftMe+4 do ShiftM+8 bez wpływania na wartość w ShiftMe+4.Druga instrukcja shld przesuwa bity od ShiftMe do ShiftMe+4.W końcu, instrukcja shl przesuwa mniej znaczące podwójne słowo o stosowną wielkość. Są dwie ważne rzeczy do odnotowania o tym kodzie. Po pierwsze, w odróżnieniu od innych operacji przesunięcia w lewo o podwyższonej dokładności, ta sekwencja pracuje od bardziej znaczącego podwójnego słowa do mniej znaczącego słowa. Po drugie, flaga przeniesienia nie zawiera przeniesienia z bardziej znaczącej operacji przesunięcia. Jeśli musimy zachować flagę przeniesienia w tym punkcie ,będziemy musieli przenieść na stos flagi po pierwszej instrukcji shld i ściagnąć ze stosu po instrukcji shl.

Możemy wykonać operację przesunięcia w prawo o podwyższonej dokładności stosując instrukcję shrd. Pracuje prawie w ten sam sposób jak powyższa sekwencja kodu z wyjątkiem działania od mniej znaczącego podwójnego słowa do bardziej znaczącego podwójnego słowa

---

9.3.12 OPERACJE OBROTU O PODWYŻSZONEJ DOKŁADNOŚCI

Rozszerzone operacje rcl i rcr działają w sposób prawie identyczny jak shl i shr .Na przykład dla wykonanie 48 bitowej operacji rcl i rcr, stosujemy poniższe instrukcje:

```
rcl     word ptr operand,1
rcl     word ptr operand+2, 1
rcl     word ptr operand+4, 1

rcr     word ptr operand+4, 1
rcr     word ptr operand+2, 1
rcr     word ptr operand, 1
```

Jedyna różnica pomiędzy tym kodem a kodem dla operacji przesunięcia o podwyższonej dokładności jest taka, że pierwszą instrukcją jest rcl lub rcr zamiast instrukcje shl lub shr .Wykonywanie instrukcji rol i ror o podwyższonej dokładności nie jest tak prostą operacją. Na procesorach 80386 i późniejszych, możemy zastosować instrukcje bt, shld i shrd dla łatwiejszej implementacji instrukcji ror i rol o podwyższonej dokładności. Poniższy kod pokazuje jak użyć instrukcji shld do wykonania rol o podwyższonej dokładności:

```
;Oblicza ROL EDX:EAX, 4
        mov     ebx, edx
        shld    edx,eax, 4
        shld    eax, ebx, 4
        bt      eax, 0          ;ustawienie flagi przeniesienia
```

Instrukcja ror o podwyższonej dokładności jest podobna; zapamiętajmy, że pracuje najpierw na najmniej znaczącym końcu obiektu, a na bardziej znaczącym jako ostatnim.

9.4 DZIAŁANIA NA OPERANDACH O RÓZNYCH WYMIARACH

Czasami możemy musieć obliczać jakieś wartości pary operandów, które nie są tego samego rozmiaru. Na przykład, możemy chcieć dodać słowo i podwójne słowo razem lub odjąć wartość bajtu z wartością słowa. Rozwiązanie jest proste: rozszerzenie małego operandu do rozmiaru operandu dużego a potem wykonanie operacji na dwóch rozmiarowo podobnych operandach.. Dla operandów ze znakiem ,powielimy znak mniejszego operandu do rozmiaru większego operandu; dla wartości bez znakowej, powielimy zero mniejszego operandu .Działa to dla każdej operacji, chociaż poniższy przykład demonstruje to dla operacji dodawania.

Dla rozszerzenia mniejszego operandu do wielkości dużego operandu stosujemy operację rozwinięcie znaku lub zera (w zależności czy dodajemy wartości ze znakiem czy bez znaku).Kiedy poszerzymy mniejszą wartość do rozmiaru większej, możemy kontynuować dodawanie .Rozważmy poniższy kod, który dodaje wartość bajtu do wartości słowa:

```
var1        byte        ?
var2        word        ?
```

Dodawanie bez znakowe:

```
mov     al., var1
mov     ah,0
add     ax, var2
```

Dodawanie ze znakiem:

```
mov     al., var1
cbw
add     ax, var2
```

W obu przypadkach, zmienna bajtowa została załadowana do rejestru al., rozszerzona do 16 bitów i potem dodana do operandu word. Kod ten powiedzie się rzeczywiście dobrze, jeśli możemy wybrać porządek operacji (np. dodanie wartości ośmiobitowej do wartości szesnastobitowej) Czasami ,nie możemy wyszczególnić. Być może szesnasto bitowa wartość jest już w rejestrze ax a my chcemy dodać do niej wartość ośmiobitową. Dla dodawania bez znakowego, możemy zastosować poniższy kod:

```
mov     ax, var2            ;ładujemy 16 bitową wartość do AX
-                           ;robimy jakieś inne operacje pozostawiając
-                           ;16 bitową wielkość w AX
add     al., var1           ;dodajemy wartość ośmio bitową
adc     ah, 0               ;dodajemy przeniesienie do bardziej znaczącego słowa
```

Pierwsza instrukcja add w tym przykładzie dodaje bajt var1 do mniej znaczącego bajtu wartości w akumulatorze. instrukcja adc dodaje przeniesienie z miej znaczącego bajtu do bardziej znaczącego bajtu akumulatora. Musimy być pewni, że instrukcja adc jest obecna .Jeśli ją opuścimy, możemy nie otrzymać prawidłowego wyniku.

Dodanie ośmiobitowego operandu ze znakiem do wartości szesnastobitowej ze znakiem jest trochę trudniejsze. Niestety, nie możemy dodać wartości bezpośrednio (jak powyżej) do bardziej znaczącego słowa w ax. Jest tak ponieważ bardziej znaczący rozszerzony bajt może być albo 00h albo 0FFh.Jeśli rejestr jest wolny ,najlepszą rzecz jaka to zrobi to:

```
mov     bx, ax          ,BX jest wolnym rejestrem
mov     al., var1
cbw
add     ax, bx
```

Jeśli dodatkowy rejestr nie jest wolny, możemy spróbować poniższego kodu:

```
        add     al., var1
        cmp     var1, 0
        jge     add0
        adc     ah, 0FFh
        jmp     addedFF
add0:   adc     ah, 0
addedFF:
```

Oczywiście ,jeśli inny rejestr nie jest wolny, możemy zawsze odłożyć go na stos i zapisać go podczas wykonywania operacji np.

```
push    bx
mov     bx, ax
mov     al.,var1
cbw
add     ax, bx
pop     bx
```

Inną alternatywą jest przechowanie 16 bitowej wartości z akumulatora w komórce pamięci a potem kontynuowanie jak przedtem:

```
mov     temp, ax
mov     al., var1
cbw
add     ax, temp
```

Wszystkie powyższe przykłady dodają wartość bajtową do wartości słowa .Poprzez rozszerzenie zera lub znaku mniejszego operandu do rozmiaru operandu większego, możemy łatwo dodać każde dwie różnorozmiarowe zmienne razem. Rozważmy poniższy kod, który dodaje operand bajtowy ze znakiem do podwójnego słowa ze znakiem:

```
var1    byte    ?
var2    dword   ?

        mov     al., var1
        cbw
        cwd                             ;rozszerzenie do 32 bitów w DX
        add     ax, word ptr var2
        adc     dx, word ptr var2+2
```

Oczywiście, jeśli mamy procesor 80386 lub późniejszy ,możemy zastosować poniższy kod:

```
movsx   eax, var1
```

```
        add             eax, var2
```
Przykładem bardziej odpowiednim dla 80386 jest dodawanie wartości ośmiobitowej do poczwórnego słowa (64 bity),rozważmy poniższy kod:

```
Bval    byte            -1
Qval    qword           1

        movsx           eax, Bval
        cdq
        add             eax, dword ptr Qval
        adc             edx, dword ptr Qval+4
```

9.5 IDIOMY MASZYNOWE I ARYTMETYCZNE

Idiom jest dziwactwem. Kilka operacji arytmetycznych i instrukcji 80x86 ma dziwactwa, które możemy wykorzystać kiedy piszemy kod języka asemblera. Niektórzy odnoszą się do stosowania maszynowych i arytmetycznych idiomów jako „skomplikowanego programowania”, którego powinniśmy zawsze unikać w dobrze napisanych programach. Podczas gdy mądrze jest unikać sztuczek, właśnie przez wzgląd na sztuczki, wiele maszynowych i arytmetycznych idiomów jest dobrze znanych i powszechnie znajdowanych w programach asemblerowych. Niektóre z nich rzeczywiście mogą być skomplikowane, ale większość to proste kuglarskie sztuczki. Ten tekst nawet nie może zacząć przedstawiać wszystkich idiomów ,które znajdują się dzisiaj w użyciu.; są one zbyt liczne ,a lista ich jest stale zmieniana .Niemniej jednak, jest kilka bardzo ważnych idiomów, które będziemy widzieli cały czas, wiec jest sens aby je omówić.

9.5.1 MNOŻENIE BEZ MUL I IMUL

Jeśli spojrzymy na czas wykonania dla instrukcji mnożenia, zauważymy, że czas wykonania tych instrukcji jest dosyć długi. Tylko instrukcje div i idiv wykonują się dłużej na 8086.kiedy mnożymy przez stałą, możemy uniknąć spadku wydajności instrukcji mul i imul przez użycie przesunięć, dodawań i odejmowań dla wykonania mnożenia.

Pamiętamy, że operacja shl wykonuje tą samą operację jak mnożenie wyszczególnionego operandu przez dwa. Przesunięcie w lewo o dwa bity mnoży operand przez cztery .Przesunięcie w lewo o trzy bity mnoży operand przez osiem. Ogólnie, przesunięcie operandu w lewo o n bitów mnoży go przez 2^n.Kazda wartość może być pomnożona przez jakąś stałą, używając serii przesunięć i dodawań lub przesunięć i odejmowań. .Na przykład, pomnożenie rejestru ax przez dziesięć ,potrzebujemy tylko pomnożyć go przez osiem a potem dodać dwa razy pomnożoną wartość oryginalną. To znaczy 10*ax =8*ax+2*ax.Kod który to wykonuje:

```
shl     ax, 1           ,mnożenie ax przez dwa
mov     bx, ax          ;zachowanie 2 * AX na później
shl     ax, 1           ,mnożenie ax przez cztery
shl     ax, 1           ;mnożenie AX przez osiem
add     ax, bx          ;dodanie 2*AX aby otrzymać 10*AX
```

Rejestr ax (albo inny przeznaczony do tego celu) może być pomnożony przez wartości stałe dużo szybciej stosując shl niż przez stosowanie instrukcji mul .Może się to wydawać trudne do uwierzenia ponieważ zabiera ona tylko dwie instrukcje do obliczenia wyniku:

```
mov     bx, 10
mul     bx
```

Jednak, jeśli spojrzymy na czas wykonania ,przykład z przesunięciami i dodawaniem wymaga mniej cykli zegarowych na większości procesorów z rodziny 80x86 niż instrukcja mul .Oczywiście, kod jest dłuższy (o kilka bajtów),ale poprawa wydajności jest zazwyczaj warta tego. Oczywiście na procesorach późniejszych 80x86,instrukcja mul jest trochę szybsza niż na procesorach wcześniejszych, ale schemat przesunięcie i dodawanie jest generalnie szybszy również na tych procesorach.

Możemy również zastosować odejmowanie z przesunięciem dla wykonania operacji mnożenia. Rozważmy mnożenie przez siedem:

```
mov     bx, ax          ;zachowanie AX*1
shl     ax, 1           ;AX := AX*2
shl     ax, 1           ;AX := AX*4
shl     ax, 1           ;AX := AX*8
sub     ax, bx          ;AX :=AX*7
```

To wynika bezpośrednio z faktu, że ax*7 = (ax*8)-ax

Powszechnym błędem robionym przez początkujących studentów języka asemblera jest odejmowanie lub dodawanie jeden lub dwa zamiast ax*1 lub ax*2.To poniżej nie obliczy ax*7:

```
shl     ax, 1
```

```
shl     ax, 1
shl     ax, 1
sub     ax, 1
```

Obliczymy (8*ax)-1,coś całkowicie innego (chyba, że ax = 1). Wystrzegajmy się takich pułapek, kiedy stosujemy przesunięcia, dodawanie i odejmowanie dla wykonani operacji mnożenia.

Możemy również zastosować instrukcję lea do obliczenia pewnych iloczynów na procesorze 80386 i późniejszych. Sztuczką jest zastosowanie trybu skalowanego z indeksowaniem. Poniższy przykład demonstruje kilka prostych przypadków:

```
lea     eax, [ecx][ecx]         ;EAX := ECX*2
lea     eax, [eax][eax*2]       ;EAX := EAX*3
lea     eax, [eax*4]            ;EAX := EAX*4
lea     eax, [ebx][ebx*4]       ;EAX := EBX*5
lea     eax, [eax*8]            ;EAX := EAX*8
lea     eax, [edx][edx*8]       ;EAX := EDX*9
```

9.5.2 DZIELENIE BEZ DIV I IDIV

Podobnie jak instrukcja shl może być zastosowana dla zasymulowanie mnożenia przez jakąś potęgę dwójki, tak instrukcje shr i sar mogą być zastosowane do zasymulowania dzielenia przez potęgę dwójki. Niestety, nie możemy zastosować przesunięcia, dodawania i odejmowania dla wykonania dzielenia przez dowolną stałą, tak łatwo jak można zastosować te instrukcje do wykonania operacji mnożenia.

Innym sposobem wykonania dzielenia jest użycie instrukcji mnożenia. możemy podzielić jakąś wartość przez pomnożenie przez jej odwrotność .Instrukcja mnożenia jest odrobinę szybsza niż instrukcja dzielenia; mnożenie przez odwrotność jest prawie zawsze szybsze niż dzielenie.

Teraz prawdopodobnie zastanawiamy się „jak pomnożyć przez odwrotność kiedy wartości ,którymi się Zajmujemy wszystkie są całkowite? ”Odpowiedź, oczywiście jest taka, że musimy zrobić to oszukańczo. Jeśli chcemy pomnożyć przez jedną dziesiątą, nie mam sposobu załadowania wartości 1/10 do rejestru 80x86 przed wykonaniem dzielenia .Jednakże, możemy pomnożyć 1/10 przez 10,wykonując mnożenie a potem dzieląc wynik przez 10 uzyskamy wynik końcowy .Oczywiście, to nie doprowadziłoby do niczego, faktycznie rzeczy zrobiły się gorsze ponieważ teraz robimy mnożenie przez dziesięć również jako dzielenie przez dziesięć. Jednak przypuśćmy, że mnożymy 1/10 przez 65,536(6553),wykonujemy mnożeni a potem dzielimy przez 65,536.To będzie jeszcze poprawnie wykonana operacja i ,okazuje się, jeśli prawidłowo założymy problem, możemy otrzymać operację dzielenia za darmo. Rozważmy poniższy kod, który dzieli ax przez dziesięć:

```
mov     dx, 6554        ;zaokrąglenie (65,536 / 10)
mul     dx
```

Kod ten pozostawi ax/10 w rejestrze dx.

Dla zrozumienia jak to działa, rozpatrzmy co wydarzy się ,kiedy pomnożymy ax przez 65,536 (10000h).Po prostu ax jest przenoszone do dx i ustawiane na zero. Pomnożenie przez 6,554 (65,536 \dzielone przez dziesięć) wkłada ax podzielone przez 10 do rejestru dx. Ponieważ mul jest tylko nieznacznie szybsza niż div, ta technika działa trochę szybciej kiedy używamy prostego dzielenia.

Mnożenie przez odwrotność pracuje dobrze kiedy musimy dzielić przez stałą. możemy nawet zastosować ją do dzielenia przez zmienną, ale koszty obliczania odwrotności opłacą się jeśli wykonanym dzielenie wiele, wiele razy (przez tą samą wartość)

9.5.3 STOSOWANIE AND DO OBLICZANIA RESZTY

Instrukcja and może być zastosowana do szybkiego obliczania reszt z postaci:

przez. := przez. MOD 2^n

Dla obliczenia reszt stosując instrukcję and, po prostu andujemy operand z wartością 2^n-1.Na przykład, dla obliczenia ax = ax mod 8 po prostu używamy instrukcji

```
and     ax, 7
```

Dodatkowe przykłady:

```
and     ax, 3           ;AX  := AX mod 4
and     ax, 0Fh         ;AX  :=  AX mod 16
and     ax, 1Fh         ;AX  := AX mod 32
and     ax, 3Fh         ;AX  := AX mod 64
and     ax, 7Fh         ;AX  := AX mod  128
mov     ah, 0           ;AX := AX mod 256
                        ;(to samo co ax and 0FFH)
```

9.5.4 IMPLEMENTACJA LICZNIKA MODULO –n Z AND

Jeśli chcemy zaimplementować licznik zmiennej, który zlicza w górę do 2^n-1 a potem resetuje do zera, po prostu użyjemy następującego kodu:

```
inc     CounterVAr
and     CounterVAr, nBits
```

gdzie nBits jest binarną wartością zawierającą n bitów jedynek wyrównanych w liczbie do prawej .na przykład, dla stworzenie licznika ,który obraca się od zera do piętnastu, użyjemy poniższy kod

```
inc     CounterVAr
and     CounterVAr, 00001111b
```

9.5.5 TESTOWANIE WARTOŚCI O PODWYŻSZONEJ DOKŁADNOŚCI DLA 0FFFF..FFh

Instrukcja and może być zastosowana do szybkiego sprawdzenia wartości wielosłowa aby zobaczyć czy zawiera jedynki na wszystkich pozycjach bitów. Po prostu ładujemy pierwsze słowo do rejestru ax a potem logicznie andujemy rejestr ax ze wszystkimi pozostałymi słowami w strukturze danych. Kiedy operacja and jest skończona ,rejestr ax będzie zawierał 0FFFFh jeśli i tylko jeśli wszystkie słowa w strukturze zawierały 0FFFFh,N.p:

```
mov     ax, word ptr var
and     ax, word ptr var+2
and     ax, word ptr var+4
-
-
-
and     ax, word ptr var+n
cmp     ax, 0FFFFh
je      Is0FFFFh
```

9.5.6 OPERACJE TEST

Pamiętamy, że instrukcja test jest instrukcją and, która nie zachowuje wyniku operacji (inaczej niż ustawianie flag).Dlatego też, wiele uwag dotyczących operacji and (zwłaszcza ze względu na sposób wpływania na flagi) również dotyczy instrukcji test. Jednakże, ponieważ instrukcja test nie wpływa na operand przeznaczenia, wielokrotne testowanie bitów może być wykonywane na tej samej wartości. Rozważmy poniższy kod:

```
test    ax, 1
jnz     Bit0
test    ax, 2
jnz     Bit1
test    ax, 4
jnz     Bit3
itd.
```

Kod ten może być zastosowany do następującego po sobie testowania każdego bitu rejestrze ax (lub każdego innego operandu dla tego celu. Zauważmy, że nie możemy zastosować pary instrukcji test /cmp do testowania dla określonej wartości wewnątrz ciągu bitów (jednak możemy użyć instrukcje and /cmp).Ponieważ test nie usuwa żadnych niepotrzebnych bitów, instrukcja cmp w rzeczywistości będzie porównywała wartości oryginalne zamiast wartości usuniętych. Z tego powody, zwykle stosujemy instrukcję test aby zobaczyć czy pojedynczy bit jest ustawiony lub czy jeden lub więcej bitów z grupy bitów są ustawione. Oczywiście, jeśli mamy procesor 80386 lub późniejsze, możemy również użyć instrukcji bt do testowania pojedynczych bitów operandzie.

Innym ważnym zastosowaniem instrukcji test jest efektywne porównanie rejestru z zerem. Poniższa instrukcja test ustawia flagę zera jeśli i tylko jeśli ax zawiera zero

```
test    ax, ax
```

Instrukcja test jest krótsza niż

```
cmp     ax, 0
```

lub

```
cmp     eax, 0
```

chociaż nie jest lepsza niż

```
cmp     al.,0
```

Zauważmy, że możemy zastosować instrukcje and i or do testowania dla zera w sposób identyczny jak test. Jednakże, na procesorach potokowych, takich jak 80486 i Pentium, przy instrukcji test jest mniejsze prawdopodobieństwo do stworzenia ryzyka ponieważ nie przechowuje wyniku w swoim rejestrze przeznaczenia.

9.5.7 TESTOWANIE ZNAKÓW INSTRUKCJĄ XOR

Pamiętamy ból związany z operacją mnożenia ze znakiem o zwielokrotnionej precyzji? Potrzebujemy określić znak wyniku, bierzemy wartości absolutne operandów, potem je mnożymy, a potem poprawiamy znak wyniku na określony przed operacją mnożenia. Znak iloczynu dwóch liczb jest po prostu exclucsive-or ich znaków przed wykonaniem mnożenia. Dlatego też, możemy użyć instrukcji xor do określenia znaku iloczynu dwóch liczb o podwyższonej dokładności. Np.

32x32 Mnożenie:

```
                mov     al., byte ptr Oprnd1+3
                xor     al., byte ptr Oprnd2+3
                mov     cl, al.                     ;zachowaj znak
;Tu robimy mnożenie (nie zapomnijmy wziąć wartości tych dwóch operandów przed wykonaniem mnożenia)
                -
                -
                -
;teraz ustalamy znak
                cmp     cl, 0                       ;sprawdzamy bit znaku
                jns     resultIsPos
;Negujemy tu iloczyn
                -
                -
                -
ResultIs Pos:
```

9.6 OPERACJE MASKOWANIA

Maska jest wartością używaną do wymuszenia pewnych bitów na zero lub jeden wewnątrz jakiejś innej wartości. Maska typowo wpływa na pewne bity w operandzie i pozostawiają inne bity nienaruszone. odpowiednie zastosowanie maski pozwala nam wyekstrahować bity z wartości, wprowadzić bity do wartości i zapakować i wypakować upakowany typ danych. Poniższa sekcja opisuje te operacje szczegółowo

---

9.6.1 OPERACJE MASKOWANIA Z INSTRUKCJĄ AND

Jeśli spojrzymy na tablicę prawdy dla operacji and w Rozdziale Pierwszym, zauważymy, że jeśli ustalimy operand na zero, wynik jest zawsze zerem. Jeśli ustawimy ten operand na jeden, wynik jest zawsze wartością drugiego operandu, Możemy zastosować tą właściwość instrukcji and do selektywnego wymuszania pewnych bitów na zero w wartości bez wpływania na pozostałe bity. Nazywa się to maskowaniem bitów.

Dla przykładu rozpatrzymy kody ASCII dla cyfr „0”..”9”.Ich kody są odpowiednio z zakresu 30h..39h.Aby skonwertować cyfry ASCII do ich odpowiednich wartości numerycznych musimy odjąć30h od kodu ASCII. Jest to łatwe do wykonania poprzez logiczne dodanie wartości 0Fh.To ustawia wszystko na zero, ale cztery mniej znaczące bity tworzą wartość liczbową. Może zastosujemy instrukcje odejmowania ,ale większość ludzi do tego celu stosuje instrukcję and.

9.6.2 OPERACJE MASKOWANIA Z INSTRUKCJĄ OR

Podobnie jak możemy zastosować instrukcję and do wymuszenia wybranych bitów na zero, możemy użyć instrukcji or do wymuszenia wybranych bitów na jeden.

Pamiętamy operację maskowania popisaną wcześniej przy instrukcji and? W tamtym przykładzie chcieliśmy skonwertować kod ASCII do cyfr jako jej liczbowego ekwiwalentu .Możemy zastosować instrukcję or dla odwrócenia tego procesu. To znaczy ,konwertujemy wartość liczbową z zakresu 0..9 do kodu ASCII odpowiadającemu cyfrze tj. ‘0’..’9’.Zrobimy to orując logicznie wyszczególnioną wartość liczbową z 30h.

9.7 PAKOWANIE I WYPAKOWYWANIE TYPÓW DANYCH

Jednym z podstawowych zastosowań instrukcji przesunięcia i obrotu jest pakowanie i wypakowywanie danych. Typy danych bajtu i słowa są wybierane dużo częściej niż inne ponieważ 80x86 wspiera te dwa rozmiary danych sprzętowo. jeśli nie potrzebujemy dokładnie ośmiu lub 16 bitów, stosowanie bajtu lub słowa do przechowywania danych może być rozrzutnością. Poprzez upakowanie danych możemy zredukować pamięć wymaganą dla naszej danej poprzez wstawienie dwóch lub więcej wartości do pojedynczego bajtu lub słowa .Kosztem takiej redukcji w pamięci jest niższa wydajność. Zabiera czas pakowanie i wypakowywanie danych .Pomimo to, dla aplikacji, dla których szybkość nie jest krytyczna (lub dla tych części aplikacji, dla których szybkość nie jest krytyczna),oszczędność pamięci może uzasadniać zastosowanie danych upakowanych.

Typ danych, który oferuje największe oszczędności kiedy stosujemy technikę pakowania, jest boolowski typ danych. Do przedstawienia prawdy lub fałszu wymaga pojedynczego bitu. Dlatego też, osiem różnych wartości boolowskich może być upakowanych w pojedynczym bajcie. To przedstawia współczynnik

kompresji 8:1,dlatego upakowana tablica wartości boolowskich wymaga tylko jedną ósmą miejsca tablicy nie upakowanej(gdzie każda boolowska zmienna zużywa jeden bajt).Na przykład pascalowska tablica

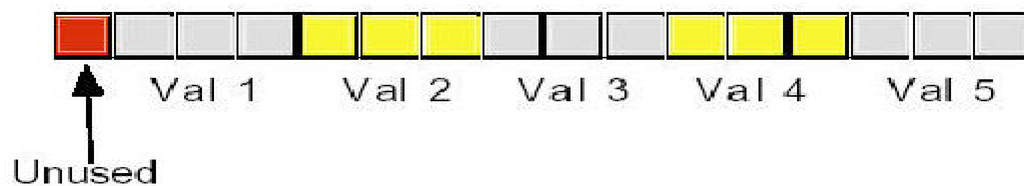
B;packed arreay[0..31] of boolean

Wymaga tylko czterech bajtów, kiedy pakujemy jedną wartość na bit. Kiedy pakujemy jedną wartość na bajt, tablica ta wymaga 32 bajtów.

Zajęcie się spakowaną tablicą boolowską wymaga dwóch operacji. Będziemy musieli wprowadzić wartość do pakowanej zmiennej (często zwanej polem upakowania) i będziemy musieli wyciągnąć wartość z pola upakowania .

Aby wprowadzić wartości do upakowanej tablicy boolowskiej musimy ustawić bit źródłowy na jego pozycji w operandzie przeznaczenia a potem przechować ten bit w operandzie przeznaczenia .Możemy to zrobić sekwencją instrukcji and, or i przesunięć. Pierwszym krokiem jest zamaskowanie odpowiednich bitów w operandzie przeznaczenia. Do tego celu używamy instrukcję and. Potem operand źródłowy jest przesuwany, żeby był ustawiony na pozycji przeznaczenia, w końcu operand źródłowy jest orowany z operandem przeznaczenia. Na przykład, jeśli chcemy wstawić bit zero rejestru ax do bitu pięć rejestru cx, zastosujemy poniższy kod:

```
and     cl, 0DFh        ;zerujemy bit pięć (bit przeznaczenia)
and     al.,1           ;zeruje wszystkie bity za wyjątkiem bitu źródłowego
ror     al., 1          ;przesuwa do bitu 7
shr     al., 1          ;przesuwa do bitu 6
shr     al., 1          ;przesuwa do bitu 5
or      cl, al.
```



Rysunek 8.4 Dane upakowane

Kod ten jest nieco skomplikowany .Obraca dane w prawo zamiast przesuwać je w lewo ponieważ wymaga to mniej instrukcji przesunięć i obrotów.

Aby wyciągnąć wartość boolowską, po prostu odwracamy ten proces. Po pierwsze, przesuwamy żądany bit do bitu zero a potem maskujemy wszystkie inne bity. Na przykład, dla wyciągnięcia danych z bitu pięć rejestru cx, pozostawiamy pojedynczą wartość boolowską w bicie zero rejestru ax, stosujemy poniższy kod:

```
mov     al. , cl
shl     al., 1          ;Bit 5 do bitu 6
shl     al., 1          ;Bit 6 do bitu 7
rol     al., 1          ;Bit 7 do bitu 0
and     ax, 1           ;zerujemy wszystkie bity  z wyjątkiem zero
```

Dla testowania zmiennych boolowskich w upakowanej tablicy, nie potrzebujemy wyciągać bitu a potem go testować ,możemy przetestować go na miejscu. Na przykład dla przetestowania wartości w bicie pięć, dla sprawdzenia czy jest tam zero czy jeden ,zastosujemy poniższy kod:

```
test    cl, 00100000b
jnz     BitIsSet
```

Inne typy danych upakowanych mogą być obsługiwane w podobny sposób z wyjątkiem, kiedy musimy pracować z dwoma lub więcej bitami. Na przykład przypuśćmy, że upakowaliśmy pięć różnych trzybitowych pól do sześciobitowej wartości jak pokazano na rysunku 8.4.

Jeśli rejestr ax zawiera dane pakowane do value3,możemy użyć poniższego kodu do wprowadzenia tej danej do pola trzy:

```
mov     ah, al.         ;robi shl przez 8
shr     ax, 1           ;repozycjonowanie do bitów 6..8
shr     ax, 1
and     ax, 11100000b   ;usunięcie niepożądanych bitów
and     DATA, 0FE3Fh    ;ustawienie żądanego pola na zero
or      DATA, ax        ;przyłączenie nowej danej do pola
```

wyłuskanie jest dokonywane w podobny sposób Najpierw usuwamy niepotrzebne bity a potem uzyskujemy wynik:

```
mov     ax, DATA
```

```
and     ax, 1Ch
shr     ax, 1
shr     ax, 1
shr     ax, 1
shr     ax, 1
shr     ax, 1
shr     ax, 1
```

Kod ten może być poprawiony poprzez zastosowanie poniższej sekwencji kodu:

```
mov     ax, DATA
shl     ax, 1
shl     ax, 1
mov     al. Ah
and     ax, 07h
```

Dodatkowe zastosowanie danych upakowanych będzie zgłębiane przez całą książę.

9.8 TABLICE

Termin „tablice" ma różne znaczenia dla różnych programistów .Dla większości programistów asemblerowych, tablica nie jest niczym więcej niż tablicą, która jest inicjowana jakąś daną. Programista asemblerowy często stosuje tablice do obliczania złożonych lub inaczej wolnych funkcji. Bardzo wiele języków wysokiego poziomu (np SNOBOLA4 i Icon) bezpośrednio wspierają tablicowy typ danych. Tablice w tych językach są zasadniczo tablicami, do elementów których możemy uzyskać dostęp z wartościami nie całkowitymi (np. zmienno przecinkowymi, ciągami lub innymi typami danych).W sekcji tej ,zaadoptujemy spojrzenie programistów asemblerowych na tablice.

Tablica jest tablicą zawierającą preinicjowane wartości, które nie zmieniają się podczas wykonywania programu. Tablica może być porównana do tablicy w ten sam sposób, jak stała całkowita może być porównana do zmiennej całkowitej .W asemblerze, możemy zastosować tablice do różnych celów; obliczania funkcji, sterownia przepływem danych .Ogólnie, tablice dostarczają szybkiego mechanizmu dla wykonani jakichś operacji kosztem przestrzeni w naszym programie (dodatkowa przestrzeń przechowuje dane tablicowe).W poniższej sekcji, zgłębimy możliwości zastosowania tablic w programach asemblerowych.

9.8.1 OBLICZANIE FUNKCJI PRZEZ PRZEKSZTAŁCENIE TABLICOWE

Tablice mogą robić różne rzeczy w asemblerze. W HLL'ach, takich jak Pascal, łatwo jest w rzeczywistości stworzyć formułę, która oblicza jakąś wartość .Proste wyrażenie arytmetyczne jest odpowiednikiem znacznej ilości kodu języka asemblerowego 80x86.Programiści języka asemblera mają tendencję do obliczania wielu wartości poprzez przekształcenia tablicowe zamiast przez wykonanie jakiejś funkcji ..Ma to zaletę bycia łatwiejszym i często również bardziej wydajnym. Rozważmy poniższą instrukcję pascalowską:

```
if (znak >= 'a') and (znak <= 'z') then znak := chr(ord(znak) – 32);
```

Ta Pascalowska instrukcja if konwertuje zmienną znak z małej litery do dużej litery jeśli znak jest z zakres 'a'..'z'.Kod asemblerowy 80x86,który wykonuje to samo jest taki:

```
            mov     al., znak
            cmp     al., 'a'
            jb      NowLower
            cmp     al., 'z'
            ja      NotLower
            and     al., 05fh               ;tak sama operacja jak SUB AL., 32
NotLower:   mov     znak, al.
```

Możemy schować ten kod w pętli zagnieżdżonej, jednak będzie trudno uzyskać poprawę szybkości kodu bez użycia przekształcenia tablicowego. Stosowanie przekształcenie tablicowe pozwala nam na zredukowanie tej sekwencji kodu tylko do czterech instrukcji:

```
mov     al., znak
lea     bx, CnvrtLower
xlat
mov     znak, al.
```

CnvrtLower jest 256 bajtową tablicą, która zawiera wartości 0..60h pod indeksami 0..60h,41h..5Ah pod indeksami 61h..7Ah i 7Bh..0FFh.Często,stosujac to udogodnienie przekształcenia tablicowego zwiększymy szybkość naszego kodu.

Ponieważ zwiększa się złożoność funkcji, korzyści z metody przekształcenia tablicowego wzrastają gwałtownie. Kiedy prawie wcale nie stosujemy tablicy wyszukiwań dla konwersji małych liter na duże ,rozpatrzmy co się stanie jeśli chcemy zamienić przypadek:

Przez obliczenie:

```
                mov     al., znak
                cmp     al., 'a'
                jb      NotLower
                cmp     al., 'z'
                ja      Notlower
                and     al., 05fh
                jmp     ConvertDone
NotLower:       cmp     al., 'A'
                jb      ConvertDone
                cmp     al., 'Z'
                ja      ConvertDone
                or      al., 20h
ConvertDone:
                mov     znak, a;
```

Kod przekształcenia tablicowego oblicza tą sama funkcję tak:

```
                mov     al., znak
                lea     bx, SwapUl
                xlat
                mov     znak, al.
```

Jak widać, kiedy obliczamy funkcję przez przekształcenie tablicowe, bez względu jak jest funkcja ,zmienia się tablica, nie kod wykonujący wyszukiwanie.

Przekształcenie tablicowe cierpi tylko z jednego głównego powodu – funkcje obliczane przez przekształcenia tablicowa mają ograniczony zakres działania. Zakresem działania funkcji jest zbiór możliwych wartości wejściowych (parametry),które akceptuje. Na przykład, powyższa funkcja konwersji duże/ małe litery ma 256 znakowy zbiór znaków ASCII jako swój zakres działania.

Funkcje takie jak SIN czy COS akceptują zbiór liczb rzeczywistych jako możliwe wartości wejściowe. Jasne ,że zakres działania dla SIN i COS jest dużo większy niż dla funkcji konwersji duże/małe litery. Jeśli mamy zamiar zrobić obliczenia przez przekształcenie tablicowe ,musimy ograniczyć zakres działania funkcji do małego zbioru. Jest tak ponieważ każdy element z zakresu działania funkcji wymaga wejścia w tablicy wyszukiwań. Nie będziesz uważał za bardzo praktyczną implementację funkcji która korzysta z przekształceń tablicowych ,której zakresem działania jest zbiór liczb rzeczywistych.

Większość tablic wyszukiwań jest całkiem małych, zazwyczaj 10 do 128 wejść. Rzadko tablica przekształceń wzrasta poza 1000 wejść. Większość programistów nie ma cierpliwości do tworzenia (i weryfikacji poprawności) 1000 wejść do tablicy.

Innym ograniczeniem funkcji opartych o tablice wyszukiwań jest to, że elementy w zakresie działania funkcji muszą być dość przyległe. Przekształcenie tablicowe pobiera wartość wejściową dla funkcji, używając tej wartości wejściowej jako indeksu tablicy i zwraca wartość do tego wejścia w tablicy. Jeśli nie przekazujemy funkcji żadnej wartości innej niż 0,100,1000 i 10 000,będzie się wydawała idealnym kandydatem do implementacji przez przekształcenie tablicowe, jej zakres działania składa się tylko z czterech pozycji. Jednakże ,tablica w rzeczywistości będzie wymagała 10 001 różnych elementów należących do zakresu wartości wejściowych. Dlatego też, nie możemy sprawnie tworzyć takiej funkcji przez przekształcenie tablicowe. W całej sekcji o tablicach ,zakładamy ,że zakres działania funkcji jest dosyć ciągłym zbiorem wartości.

Najlepszymi funkcjami, które mogą być implementowana poprzez przekształcenia tablicowe są te, których zakres działania jest zawsze 0..255 (lub jakiś podzbiór tego zakresu).Takie funkcje są wydajniej implementowane na 80x86 przez instrukcje XLAT. Podprogram konwersji duże/małe litery przedstawiony wcześniej jest dobrym przykładem takiej funkcji. Każda funkcja w tej klasie (te których zakres jest od 0 do 255) może być obliczana przez zastosowanie tak samo dwóch instrukcji (lea bx, table / xlat).jedyną rzeczą która zawsze się zmienia jest tablica wyszukiwań.

Instrukcja xlat nie może być (dogodnie) zastosowana do obliczenia wartości funkcji ,której zakres działania lub zakres wychodzi poza zakres 0..255.Są trzy sytuacje do rozpatrzenia:

- Obszar stosowania wychodzi poza 0..255 ale zakres jest wewnątrz 0..255
- Obszar stosowani jest wewnątrz0..255 ale zakres jest poza 0.255, i
- Oba, obszar stosowania i zakres funkcji mają wartości poza 0..255

Rozpatrzymy każdy z tych przypadków z osobna

Jeśli obszar stosowania funkcji wychodzi poza 0.255 ale zakres funkcji mieści się wewnątrz tego zbioru wartości, nasza tablica wyszukiwań będzie wymagała więcej niż 256 wejść bale możemy przedstawić

każde wejście pojedynczym bajtem. Dlatego też, tablica wyszukiwań może być tablicą bajtów. Obok połączenia wymagającego instrukcji xlat ,funkcje mieszczące się w tej klasie są bardziej wydajne. Poniższa Pascalowska funkcja

```
B: = Func(X);
```

Gdzie Func to

```
function Func (X:word):byte
```

składa się z następującego kodu 80x86

```
mov     bx, X
mov     al., FuncTable  [bx]
mov     B, al.
```

Kod ten ładuje parametr funkcji do bx ,stosując tą wartość (z zakresu 0..??),jako indeks do tablicy FuncTable, pobierając bajt spod tej lokacji i przechowując wynik w B. Oczywiście tablica musi zawierać poprawne wejścia dla każdej możliwej wartości X. Na przykład przypuśćmy, że chcieliśmy zmapować pozycję kursora na wyświetlaczu w zakresie 0.1999 (są 2 000 pozycje znaki na wyświetlaczu 80x25) do koordynaty X lub Y na ekranie. Możemy łatwo obliczyć koordynatę X przez funkcję X:= Posn mod 80 i koordynatę Y z formuły Y:= Posn div 80 (gdzie Posn jest pozycją kursora na ekranie)Jest to łatwe do obliczenia stosując kod 80x86:

```
mov     bl, 80
mov     ax, Posn
div     bx
; X jest teraz w AH,Y jest w AL.
```

Jednakże, instrukcja div na 80x86 jest bardzo wolna. Jeśli musimy robić to obliczenie dla każdego znaku napisanego na ekranie, poważnie zmniejszymy szybkość kodu naszego wyświetlacza .poniższy kod, który realizuje te dwie funkcje przez przekształcenie tablicowe, poprawi znacznie wydajność naszego kodu :

```
mov     bx, Posn
mov     al., Ycoord[bx]
mov     ah, Xcoord[bx]
```

Jeśli obszar stosowania funkcji jest wewnątrz 0.255 ale zakres jest poza tym zbiorem, tablica wyszukiwań będzie zawierała 256 lub mniej wejść ale każde wejście będzie wymagało dwóch lub więcej bajtów. Jeśli oba zakres i obszar stosowania funkcji są poza 0.255,każde wejście będzie wymagało dwóch lub więcej bajtów a tablica będzie zawierała więcej niż 256 wejść.

Przypomnijmy sobie z Rozdziału Czwartego formułę dla indeksowania jednowymiarowej tablicy (której tablica jest specjalnym przypadkiem):

Adres: = Baza + indeks* rozmiar

Jeśli elementy z zakresu funkcji wymagają dwóch bajtów, wtedy indeks musi być pomnożony przez dwa przed indeksowaniem tablicy. Podobnie, jeśli każde wejście wymaga trzech ,czterech lub więcej bajtów ,indeks musi być pomnożony przez rozmiar każdego wejścia tablicy przez zastosowaniem jako indeksu do tablicy .Na przykład przypuśćmy, że mamy funkcję F(x),zdefiniowaną przez (pseudo) pascalowską deklarację:

```
function F(x:0..999):word
```

Możemy łatwo stworzyć ta funkcję stosując poniższy kod 80x86 (i oczywiście, odpowiednia tablicę):

```
mov     bx, X           ;pobranie wartości wejściowej funkcji i konwersja
shl     bx, 1           ; indeksu słowa do F
mov     ax, F[bx]
```

Instrukcja shl mnoży indeks przez dwa, dostarczając właściwego indeksu do tablicy której elementami są słowa.

Każda funkcja, której obszar stosowania jest mały i głownie zwartym jest dobrym kandydatem dla obliczenia przez przekształcenia tablicowe. W takim przypadku, nie- zwarte obszary stosowania są również do przyjęcia, tak długo jak obszar stosowania może być sprowadzony do właściwego zbioru wartości. Takie operacje są nazywane uzależnianiem i są tematem następnej sekcji.

9.8.2 UZALEŻNIANIE OBSZARÓW STOSOWANIA

Uzależnianie obszarów stosowania jest pobraniem zbioru wartości w obszarze stosowani funkcji i przetworzenie ich ,tak, żeby były bardziej akceptowalne jako dane wejściowe do tej funkcji .rozważmy poniższą funkcję:

$$\sin x = \langle \sin x | x \in [-2\pi, 2\pi] \rangle$$

Mówi ona, że (komputerowa) funkcja SIN(x) jest odpowiednikiem (matematycznej) funkcji sin x gdzie

$$-2\pi \leq x \leq 2\pi$$

Jak wszyscy wiemy, sinus jest funkcją cykliczną, która akceptuje każdą wejściową wartość rzeczywistą Formuła stosuje obliczenie sinusa, jednak, tylko akceptuje mały zbiór tych wartości.

To ograniczenie zakresu nie przedstawia rzeczywistego problemu ,poprzez proste obliczenie SIN(X mod (2*pi)) możemy obliczyć sinus każdej wartości wejściowej. Modyfikacja wartości wejściowej tak ,żebyśmy mogli łatwo obliczyć funkcję jest nazywana uzależnieniem wartości wejściowej. W powyższym przykładzie obliczyliśmy X mod 2*pi i stosując wynik jako daną wejściowa funkcji sin. Zaokrągla X do obszaru stosowania sin bez wpływania na wynik .Możemy zastosować uzależnienie wejścia ,możemy również zastosować przekształcenie tablicowe .Faktycznie skalowanie indeksu posługując się wejściami słowa jest formą uzależnienia wejścia .rozważmy poniższą funkcję Pascalowską:

```
function val (x:word):word; begin
        case  x of
                0: val :=1;
                1: val :=1;
                2: val :=4;
                3: val := 27;
                4: val := 256;
                inne val :=0;
        end;
end;
```

Funkcja ta oblicza jakąś wartość dla x z zakresu 0..4 i zwraca zero jeśli x jest poza zakresem. Ponieważ x może przybrać 65 536 różnych wartości (będących 16 bitowym słowem).stworzenie tablicy zawierającej 65 536 słów gdzie tylko pierwsze pięć wejść jest nie zerowych ,będzie całkiem nieekonomiczne. Jednakże możemy jeszcze obliczyć tą funkcję stosując przekształcenie tablicowe jeśli zastosujemy uzależnienie wejścia. poniższy kod asemblerowy przedstawia tą zasadę:

```
                xor     ax, ax                  ;AX = 0, zakładamy X > 4
                mov     bx,  x
                cmp     bx, 4
                ja      ItsZero
                shl     bx, 1
                mov     ax, val [bx]
ItsZero:
```

Kod ten sprawdza czy x jest poza zakresem 0..4.jeśli tak, fizycznie ustawia ax na zero, w innym przypadku odszukuje wartość funkcji w tablicy val. z uzależnieniem wejścia ,możemy zaimplementować kilka funkcji ,które w innym wypadku byłyby niepraktyczne do zrobienia przez przekształcenie tablicowe.

9.8.3 GENEROWANIE TABLIC

Jednym sporym problemem, z zastosowaniem przekształcenia tablicowego jest tworzenie tablicy. jest to szczególnie prawdziwe jeśli jest duża liczba wejść w tablicy. Obliczanie danych do umieszczenia w tablicy, potem mozolne wprowadzanie danych, a w końcu sprawdzanie tych danych aby upewnić się, że są poprawne, jest czasochłonnym i nudnym procesem. Dla różnych tablic jest lepszy sposób – użyjemy komputera do wygenerowania tablicy dla nas. Przykład jest dużo lepszy iż tylko opis .Rozważmy poniższą zmodyfikowaną funkcję sinus:

$$(\sin x) \times r = \left\langle \frac{r \times (1000 \times \sin x)}{1000} \right| x \in [0,$$

To świadczy, że x jest wartością całkowitą z zakresu 0..359 i r jest wartością całkowitą. Komputer może łatwo obliczyć to z poniższego kodu:

```
                mov     bx, X
                shl     bx, 1
                mov     ax, Sinus [bx]          ;pobranie SIN(X)*1000
                mov     bx, R                   ;obliczanie R*(SIN(X)*1000)
                mul     bx
                mov     bx, 1000                ;obliczanie (R*(SIN(X)*1000))/ 1000
                div     bx
```

Zauważmy, że mnożenie całkowite i dzielenie nie są łączne .nie możemy usunąć mnożenia przez 1000 i dzielenia przez 1000 ponieważ wywołałoby to anulowanie jedno drugiego .Co więcej ten kod musi obliczyć tą funkcję dokładnie w ten sposób. To co otrzymujemy na koniec tej funkcji jest tablica zawierająca 360 różnych

wartości odpowiadających sinusowi kąta (w stopniach razy 1000.Wprowadzanie tablicy do programu asemblerowego zawierającą takiej wartości jest niezmiernie nudne i prawdopodobnie popełnimy kilka błędów wprowadzając i weryfikując te dane .Jednak możemy mieć program generujący taką tablicę dla nas. Rozważmy poniższy program Turbo Pascala:

```
program maketable;
var     i:integer;
        r:integer;
        f::text
begin
        assign (f,'sinus.asm');
        rewrite (f);
        for i := 0 to 359 do begin
                r:= round(sin(I*2.0*pi / 360.0)*1000.0);
                if (i mod 8) = 0 then begin
                        writeln(f)
                        write (f, 'dw', r);
                end
                else write(f,',',r)
        end;
        close(f);
end.
```

Program ten tworzy poniższe dane wyjściowe:

```
dw 0,17,35,52,70,87,105,122
dw 139,156,174,191,208,225,242,259
dw 276,292,309,326,342,358,375,391
dw 407,423,438,454,469,485,500,515
dw 530,545,559,574,588,602,616,629
dw 643,656,669,682,695,707,719,731
dw 743,755,766,777,788,799,809,819
dw 829,839,848,857,866,875,883,891
dw 899,906,914,921,927,934,940,946
dw 951,956,961,966,970,974,978,982
dw 985,988,990,993,995,996,998,999
dw 999,1000,1000,1000,999,999,998,996
dw 866,857,848,839,829,819,809,799
dw 788,777,766,755,743,731,719,707
dw 695,682,669,656,643,629,616,602
dw 588,574,559,545,530,515,500,485
dw 469,454,438,423,407,391,375,358
dw 342,326,309,292,276,259,242,225
dw 208,191,174,156,139,122,105,87
dw 70,52,35,17,0,-17,-35,-52
dw -70,-87,-105,-122,-139,-156,-174,-191
dw -208,-225,-242,-259,-276,-292,-309,-326
dw -342,-358,-375,-391,-407,-423,-438,-454
dw -469,-485,-500,-515,-530,-545,-559,-574
dw -588,-602,-616,-629,-643,-656,-669,-682
dw -695,-707,-719,-731,-743,-755,-766,-777
dw -788,-799,-809,-819,-829,-839,-848,-857
dw -866,-875,-883,-891,-899,-906,-914,-921
dw -927,-934,-940,-946,-951,-956,-961,-966
dw -970,-974,-978,-982,-985,-988,-990,-993
dw -995,-996,-998,-999,-999,-1000,-1000,-1000
dw -999,-999,-998,-996,-995,-993,-990,-988
dw -985,-982,-978,-974,-970,-966,-961,-956
dw -951,-946,-940,-934,-927,-921,-914,-906
dw -899,-891,-883,-875,-866,-857,-848,-839
dw -829,-819,-809,-799,-788,-777,-766,-755
dw -743,-731,-719,-707,-695,-682,-669,-656
dw -643,-629,-616,-602,-588,-574,-559,-545
dw -530,-515,-500,-485,-469,-454,-438,-423
dw -407,-391,-375,-358,-342,-326,-309,-292
dw -276,-259,-242,-225,-208,-191,-174,-156
dw -139,-122,-105,-87,-70,-52,-35,-17
```

Oczywiście jest dużo łatwiej napisać program w Turbo Pascalu, który generuje te dane niż wprowadzać (i weryfikować) te dane ręcznie. Ten krótki przykład pokazuje jak może być użyteczny Pascal dla programisty asemblerowego.

9.12 PODSUMOWANIE

Rozdział ten omówił arytmetyczne i logiczne operacje na CPU 80x86.Przedstawił instrukcje i techniki konieczne do wykonania arytmetyki całkowitej na podobną modę jak języki wysokiego poziomu. Rozdział ten również omówił operacje o zwielokrotnionej precyzji, jak wykonać operacje arytmetyczne stosując nie arytmetyczne instrukcje i jak użyć instrukcji arytmetycznych do wykonania operacji nie arytmetycznych.

Wyrażenia arytmetyczne są dużo prostsze w językach wysokiego poziomu niż w języku asemblera. Istotnie ,pierwotnym celem Języka FORTRAN było dostarczanie FORmula TRANSlator (Tłumacz Formuł) dla wyrażeń arytmetycznych. Chociaż zabiera trochę więcej wysiłku konwertowanie formuł arytmetycznych na asembler niż powiedzmy, Pascalowi ,tak długo jak będziemy postępować według bardzo prostych reguł, konwersja nie będzie trudna. Po opis krok po kroku zobacz:

*Wyrażenia arytmetyczne
*Proste przypisania
*Proste wyrażenia
*Wyrażenia złożone
*Operatory przemienności
*Wyrażenia Logiczne (Boolowskie)

Jedną dużą zaletą języka asemblera jest to ,że jest łatwo wykonać prawie nie ograniczone dokładnością operacje arytmetyczne i logiczne. Rozdział ten opisuje jak wykonać operacje o podwyższonej dokładności dla większości powszechnych działań. Po komplet instrukcji zobacz;

*Operacje o zwielokrotnionej dokładności
*Operacje dodawania o zwielokrotnionej dokładności
*Operacje odejmowania o zwielokrotnionej dokładności
*Porównania o podwyższonej dokładności
*Mnożenie o podwyższonej dokładności
*Dzielenie o podwyższonej dokładności
*Operacje NEG o podwyższonej dokładności
*Operacje AND o podwyższonej dokładności
*Operacje OR o podwyższonej dokładności
*Operacje NOT o podwyższonej dokładności
*Operacje przesunięcia o podwyższonej dokładności
*operacje obrotu o podwyższonej dokładności

W pewnym momencie możemy musieć działać na dwóch operandach, które są różnych typów .Na przykład, możemy musieć dodać wartość bajtową z wartością słowa. Ogólną ideą jest poszerzenie mniejszego operandu tak, żeby był tego samego rozmiaru co operand większy a potem obliczyć wynik tych operandów .Po więcej szczegółów zajrzyj

*Operacje na operandach o różnych rozmiarach

Chociaż zbiór instrukcji 80x86 dostarcza prosty sposób osiągania wielu zadań, możemy często wykorzystać różne idiomy w zbiorze instrukcji lub w związku z pewnymi operacjami arytmetycznymi tworzyć kod, który jest szybszy lub krótszy niż kod oczywisty. Rozdział ten wprowadził kilka tych idiomów.

*Idiomy Maszynowe i arytmetyczne
*Mnożenie bez MUL i IMUL
*Dzielenie bez DIV i IDIV
*Stosowanie AND do obliczania reszty
*Implementowanie licznika Modulo-n z AND
*Testowanie wartości o podwyższonej dokładności dla 0FFFF..FFh
*Operacje0 TEST
*Testowanie znaków instrukcją XOR

Dla manipulowania danymi upakowanymi potrzebujemy zdolności do wyciągania pola z upakowanego rekordu i wprowadzania pola do upakowanego rekordu. Możemy użyć logicznych instrukcji and i or do maskowania pól, którymi chcemy manipulować; możemy zastosować instrukcje shl i shr do pozycjonowania danych do ich właściwych pozycji przed wprowadzeniem lub po wyciągnięciu danej. po naukę jak to robić zajrzyj

*Operacje maskowania
*Operacje maskowania instrukcją AND
*Operacje maskowania instrukcją OR
*Upakowane i rozpakowane typy danych

9.13 PYTANIA

1) Opisz jak możemy dodać zmienną bez znakowego słowa do zmiennej bez znakowego bajtu, tworząc wynik bajtowy. Wyjaśnij okoliczności wystąpienia błędu i jak poradzić sobie z nim
2) Odpowiedz na pytanie jeden dla wartości ze znakiem
3) Zakładamy, że var1 jest słowem a var2 i var3 są podwójnymi słowami .Jaki jest kod asemblerowy 80x86,który doda var1 do var2 pozostawiając wynik w var 3 jeśli:
   a) var1,var2 i var3 są wartościami bez znaku
   b) var1,var2 i var 3 są wartościami ze znakiem

4) „ADD BX, 4" jest bardziej wydajne niż :LEA BX,4[BX].Podaj przykład instrukcji LEA, która jest bardziej wydajna niż odpowiadająca jej instrukcja ADD
5) Dostarcz pojedynczej instrukcji LEA 80386,która mnoży EAX przez pięć
6) Zakładamy, że VAR1 i VAR2 są 32 bitowymi zmiennymi zadeklarowanymi z pseudo-opcodem DWORD. Napisz sekwencję kodu, która testuje co następuje:
   a)VAR1 = VAR2
   b) VAR1 <> VAR2
   c) VAR1 < VAR2 (wersje ze znakiem i bez znaku dla każdej z nich)
   d) VAR1 <= VAR2
   e) VAR1 > VAR2
   f) VAR1 >= VAR2
7) Skonwertuj poniższe wyrażenia do języka asemblera używając przesunięć, dodawań, odejmowań w miejsce mnożenia:
   a) AX*15
   b) AX*129
   c) AX*1024
   d) AX*20000
8) Jaki jest najlepszy sposób podzielenia rejestru AX przez poniższe stałe?
   a) 8 b)255 c)1024 d) 45
9) opisz jak można pomnożyć wartość ośmio bitową w AL. Przez 256 (pozostawiając wynik w AX) stosując instrukcje MOV
10) Jak można logicznie zANDować wartość w AX przez 0FFh stosując instrukcję MOV?
11) Przypuśćmy, że rejestr AX zawiera parę upakowanych wartości binarnych z najmniej znaczącymi czterema bitami zawierającymi wartość z zakresu 0..15 i 12 bardziej znaczącymi bitami zawierającymi wartość z zakresu 0.4095.teraz przypuśćmy, że chcemy zobaczyć czy porcja 12 bitów zawiera wartość 295.Wyjaśnij jak można to wykonać za pomocą dwóch instrukcji.
12) Jak można użyć instrukcję TEST (lub sekwencję instrukcji TEST) aby zobaczyć czy bit zero i cztery w rejestrze AL oba są ustawione na jeden? Jak zastosować instrukcję TEST aby zobaczyć czy jeden albo drugi bit jest ustawiony? Jak użyć instrukcję TEST aby zobaczyć czy żaden bit nie jest ustawiony?
13) Dlaczego rejestr CL nie może być użyty jako operand licznika kiedy przesuwamy operand o zwielokrotnionej precyzji. Tj. dlaczego poniższe instrukcje nie przesuną wartości w (DX,AX) trzy bity w lewo?
```
mov     cl, 3
shl     ax, cl
rcl     dx, cl
```
14) Dostarcz sekwencji instrukcji ,które wykonają operacje ROL i ROR (32 bitowe) o podwyższonej dokładności stosując tylko instrukcje 8086
15) Dostarcz sekwencji instrukcji, które implementują 64 bitową operację ROR stosując instrukcje 80386 SHRF i BT
16) Dostarcz kodu 80386 do wykonania poniższych 64 bitowych obliczeń .Zakładamy, że obliczamy X := Y op Z, z X,Y i Z zdefiniowanymi jak następuje:
```
X        dword   0, 0
Y        dword   1, 2
Z        dword   3, 4
```
   a) dodawanie b)odejmowanie c) mnożenie d) logiczne AND e) logiczne OR f) logiczne XOR
   g) negacja h) logiczne NOT