ROZDZIAŁ PIĘTNASTY: **CIĄGI ZNAKÓW I ZESTAWY ZNAKÓW**

Ciąg jest zbiorem obiektów przechowywanych w przylegających do siebie komórkach pamięci. Ciągi są zazwyczaj tablicami bajtów, słów lub (w 80386 i późniejszych procesorach) podwójnych słów Procesory z rodziny 80x86 wspierają kilka instrukcji specjalnie zaprojektowanych do kopiowania ciągów. Rozdział ten zgłębi kilka zastosowań tych instrukcji ciągów.

8088, 8086, 80186 i 80286 mogą przetwarzać dwa typy ciągów: ciągi bajtów i ciągi słów. 80386 i późniejsze procesory działają również na ciągach podwójnych słów. Mogą one przenosić ciągi, porównywać ciągi, szukać określonej wartości wewnątrz ciągu, inicjalizować ciąg stałą wartością i inne elementarne operacje na ciągach. Instrukcje ciągów 80x86 również są użyteczne dla manipulowania tablicami, tabelami i rekordami. Możemy łatwo przypisać lub porównać takie struktury danych używając instrukcji ciągów. Używając tych instrukcji możemy znacznie przyspieszyć kod działający na tablicach.

15.0 WSTĘP

Rozdział ten przedstawia przegląd operacji instrukcji ciągów 80x86. Potem omawia jak przetwarzać ciągi znaków używając tych instrukcji . W końcu, omawia instrukcje ciągów dostępne w Bibliotece Standardowej UCR. Poniższe sekcje, które mają prefiks „ •” są niezbędne. Te sekcje z „⊗” omawiają zaawansowane tematy, które możemy odłożyć na później.

- Instrukcje ciągów 80x86
- Ciągi znaków
- Funkcje ciągów znaków
- Funkcje ciągów w Bibliotece Standardowej UCR

⊗ Zastosowanie instrukcji ciągów w innych typach danych

15.1 INSTRUKCJE CIĄGÓW 80X86

Wszyscy członkowie rodziny 80x86 wspierają pięć różnych instrukcji ciągów: movs, cmps, scas, lods i stos. Są one ciągami elementarnymi ponieważ możemy zbudować większość innych operacji ciągów z tych pięciu instrukcji. Jak używać tych pięciu instrukcji , jest tematem następnych kilku sekcji.

15.1.1 JAK DZIAŁAJĄ INSTUKCJE CIĄGÓW

Instrukcje ciągów działają na blokach (tablicach o liniowej ciągłości) pamięci. Na przykład, instrukcja movs przesuwa sekwencję bajtów z jednej lokacji pamięci do innej. Instrukcja cmps porównuj dwa bloki w pamięci. Instrukcja scas przeszukuje blok pamięci pod katem określonej wartości. Te instrukcje ciągów często wymagają trzech operandów, adresu bloku przeznaczenia, adresu bloku źródłowego i (opcjonalnie) elementu zliczającego. Na przykład, kiedy używamy instrukcji movs do kopiowania ciągu, potrzebujemy adresu źródłowego i licznika (liczby elementów ciągu do przesunięcia)

W odróżnieniu od innych instrukcji które działają na pamięci, instrukcje ciągów są instrukcjami jednobajtowymi, które nie maja żadnych jasnych operandów. Argumenty dla instrukcji ciągów to:

- Rejestr si (indeks źródła)
- Rejestr di (indeks przeznaczenia
- Rejestr cx (licznik)
- Rejestr ax i
- Flaga kierunku w rejestrze FLAGS

A przykład jeden wariant instrukcji movs (przenieś ciąg) kopiuje ciąg spod adresu określonego przez ds:si do adresu określonego przez es:di, o długości cx. Podobnie instrukcja cmps porównuje ciąg wskazywany przez ds:si o długości cx z ciągiem wskazywanym przez es:di

Nie wszystkie instrukcje mają operand źródłowy i przeznaczenia (tylko movs i cmps) . Na przykład instrukcja scas (przeszukaj ciąg) porównuje wartość w akumulatorze z wartością w pamięci. Pomimo różnic, instrukcje ciągów wszystkie mają jedną rzecz wspólną – używanie ich wymaga abyśmy działali na dwóch segmentach, segmencie danych i dodatkowym segmencie.

15.1.2 PRZDROSTKI REP / REPE / REPZ I REPNZ / REPNE

Instrukcje ciągów same z siebie nie działają na ciągach danych. Instrukcja movs na przykład będzie przesuwała pojedynczy bajt, słowo lub podwójne słowo. Kiedy będzie się wykonywała, instrukcja movs zignoruje wartość z rejestru cx. Przedrostki powtórki mówią 80x86 aby wykonał wielobajtową operację na ciągach. Składnia dla przedrostków powtórki to:

Pole: Etykieta	repeat	mnemonik	argument	;komentarz
Dla MOVS:				
	rep	movs	{argumenty}	
Dla CMPS:				
	repe	cmps	{argumenty}	
	repz	cmps	{argumenty}	
	repne	cmps	{argumenty}	
	repnz	cmps	{argumenty}	
Dla SCAS:				
	repe	scas	{argumenty}	
	repz	scas	{argumenty}	
	repne	scas	{argumenty}	
	repnz	scas	{argumenty}	
Dla STOS:				
	rep	stos	{argumenty}	

Zazwyczaj nie będziemy używali przedrostków powtórzenia z instrukcją lods.

Jak widzimy, obecność przedrostków powtórzenia wprowadza nowe pole w lini źródłowej – pole przedrostka powtórzenia. Pole to pojawia si tylko w lini źródłowej zawierającej instrukcje ciągów. W naszym pliku źródłowym:

- etykieta pola powinna zawsze zaczynać się w kolumnie jeden
- pole powtórzenia powinno zaczynać się od pierwszego miejsca tabulacji , i
- pole mnemonika powinno zaczynać się od drugiego miejsca tabulacji

Kiedy określamy przedrostek powtórzenia przed instrukcją ciągu, instrukcja ta jest powtarzana cx razy. Bez tego przedrostka, instrukcja działa tylko na pojedynczym bajcie, słowie lub podwójnym słowie. Możemy użyć przedrostka powtórzenia do przetwarzania całego ciągu pojedyncza instrukcją. Możemy użyć instrukcji ciągów, bez przedrostka powtórzenia, jako elementarnych operacji na ciągach do zsyntetyzowania bardziej złożonych operacji na ciągach.

Pole operacji jest opcjonalne. Jeśli jest obecne, MASM po prostu używa go do określenia rozmiaru ciągu na jakim będzie działał. Jeśli pole operandu jest nazwą zmiennej bajtowej, instrukcja ciągu będzie działała na bajtach. Jeśli argument jest adresem słowa, instrukcja działa na słowie. Podobnie z podwójnym słowem. Jeśli pole argumentu nie jest obecne, musimy dołączyć „B”, „W” lub „D” na końcu instrukcji ciągu dla oznaczenia rozmiaru np. movsb, movsw lub movsd.

15.1.3 FLAGA KIERUNKU

Oprócz rejestrów si, di si i ax, jednym z rejestrów sterujących instrukcjami ciągów 80x86 jest rejestr flag. Ściśle, flaga kierunku w rejestrze sterującym flag, jeśli CPU przetwarza ciągi.

Jeśli flaga kierunku jest wyzerowana, CPU zwiększa si i di po operacji na każdym elemencie ciągu. Na przykład, jeśli flaga kierunku jest wyzerowana, wtedy wykonanie movs przeniesie bajt, słowo lub podwójne słowo spod ds:si do es:di i zwiększy si i di o jeden, dwa lub cztery. Kiedy wyspecyfikujemy przedrostek rep przed tą instrukcją, CPU zwiększy si i di dla każdego elementu w ciągu. Po zakończeniu, rejestry si i di będą wskazywały pierwszą pozycję poza ciągiem.

Jeśli flag kierunku jest ustawiona, wtedy 80x86 zmniejsza si i di po przetworzeniu każdego elementu ciągu. Po powtórzeniu operacji na ciągu, rejestry si i di będą wskazywały na pierwszy bajt lub słowo przed ciągiem, jeśli flaga kierunku była ustawiona.

Flaga kierunku może być ustawiona lub zerowana przy użyciu instrukcji cld (zeruj flagę kierunku) i std (ustaw flagę kierunku). Kiedy używamy tych instrukcji wewnątrz procedury, zapamiętajmy, że modyfikują one stan maszynowy. Dlatego też musimy zachować flagę kierunku podczas wykonywania tej procedury. Poniższy przykład wskazuje rodzaje problemów jakie możemy napotkać:

```
StringStuff:
                cld
        <jakieś operacje>
                call    Str2
        <jakiś operacje na ciągach wymagające D=0>
                -
                -
                -
Str2            proc    near
                Std
        <jakieś operacje na ciągach>
                ret
Str2            endp
```

Kod ten nie pracuje właściwie. Kod wywołujący zakłada, że flaga kierunku jest wyzerowana po powrocie Str2.Jednakże, nie jest to prawda. Dlatego też, operacje na ciągach wykonywane po wywołaniu Str2 nie funkcjonują poprawnie.

Jest parę sposobów zadziałania z tym problemem. Pierwszy, i prawdopodobnie najbardziej oczywisty, to zawsze wprowadzać instrukcje cld i std bezpośrednio przed wykonywaniem instrukcji na ciągach. Inną alternatywą jest zachowanie i przywrócenie flagi kierunku używając instrukcji pushf i popf. Po zastosowaniu tych dwóch technik powyższy kod będzie wyglądał jak następuje:

Zawsze wstawimy cld i std przed instrukcje ciągów:

```
StringStuff:
        cld
<jakieś operacje>
        call    Str2
        cld
<operacje wymagające D = 0>
        -
        -
        -
Str2    proc    near
        std
<jakieś operacje na ciągach>
        ret
Str2    endp
```

Zachowanie i przywrócenie rejestr flag:

```
StringStuff:
        cld
<jakieś operacje>
        call    Str2
<operacje wymagające D = 0>
```

```
                -
                -
                -
Str2            proc    near
                pushf
                std
        <jakieś operacje>
                popf
                ret
Str2            endp
```

Jeśli użyjemy instrukcji pushf i popf do zachowania i przywrócenia rejestru flag, zapamiętajmy, że zachowujmy i przywracamy wszystkie flagi. Dlatego też, taki podprogram nie może zwracać żadnych informacji we flagach. Na przykład, nie będziemy mogli zwrócić warunku błędu we fladze przeniesienia jeśli użyliśmy pushf i popf.

15.1.4 INSTRUKCJA MOVS

Instrukcja movs przybiera cztery podstawowe formy. Movs przenosi bajty, słowa lub podwójne słowa, movsb przesuwa ciąg bajtowy, movsw przenosi ciąg słów a movsd przenosi ciąg podwójnego słowa (na 80386 lub późniejszych procesorach). Te cztery instrukcje używają następującej składni:

```
{REP} MOVSB
{REP} MOVSW
{REP} MOVSD                             ;dostępna na 80386 lub późniejszych procesorach
{REP} MOVS    Przeznaczenie, Źródło
```

Instrukcja movsb (przenieś ciąg bajtowy) pobiera bajt spod adresu ds.:si, przechowuje go pod adresem es:di, a potem zwiększa lub zmniejsza rejestry si i di o jeden. Jeśli jest obecny przedrostek rep, CPU sprawdza cx aby zobaczyć czy zawiera zero. Jeśli nie, wtedy przenosi bajt z ds.:si do es:di i zmniejsza rejestr cx. Ten proces będzie się powtarzał dopóki cx nie będzie zawierał zero.

Instrukcja movsw (przenieś ciąg słów) pobiera słowo spod adresu ds.:si, przechowuje go pod adresem es:di a potem zwiększa lub zmniejsza si i di o dwa. Jeśli jest przedrostek rep, wtedy CPU powtarza tą procedurę tak długo jak jest to określone w cx.

Instrukcja movsd działa w podobny sposób na podwójnych słowach. Zwiększa lub zmniejsza si i di o cztery dla każdego przesunięcia danych.

MASM automatycznie wylicza rozmiar instrukcji movs poprzez „rzut oka” na rozmiar określonych argumentów. Jeśli zdefiniujemy dwa argumenty dyrektywą byte (lub porównywalną), wtedy MASM wygeneruje instrukcję movsb. Jeśli zadeklarujemy dwie etykiety przez word (lub porównywalne), MASM wygeneruje instrukcję movsw. Jeśli zadeklarujemy dwie etykiety jako dword, MASM wygeneruje instrukcję movsd. Asembler również sprawdzi segmenty argumentów aby zapewnić, że pasują one do aktualnie założonych (przez dyrektywę assume) rejestrów es i ds. Zawsze powinniśmy używać postaci movsb, movsw i movsd i zapomnieć o postaci movs.

Chociaż teoretycznie, forma movs wydaje się być eleganckim sposobem manipulowania instrukcjami przenoszenia ciągów, w praktyce tworzy więcej problemów niż jest to warte. Co więcej ta forma instrukcji przenoszenia ciągów sugeruje, że movs ma jawne argumenty, kiedy w rzeczywistości rejestry si i di pośrednio określają argumenty. Z tego powodu zawsze używajmy instrukcji movsb, movsw i movsd. Kiedy używamy przedrostka rep, instrukcja movsb przeniesie liczę bajtów określoną w rejestrze cx. Poniższy fragment kodu kopiuje 384 bajty z String1 do String2:

```
                cld
                lea     si, String1
                lea     di, String2
                mov     cx, 384
        rep     movsb
                -
                -
                -
String1         byte    384   dup (?)
String2         byte    384    dup (?)
```

Kod ten oczywiście zakłada, że String1 i String2 są w tym samym segmencie i oba rejestry ds. i es wskazują ten segment. Jeśli zastąpimy movsb przez movsw, wtedy powyższy kod przeniesie384 słowa (768 bajtów) zamiast 384 bajtów:

```
                cld
                lea     si, String1
                lea     di, String2
                mov     cx, 384
        rep     movsw
                -
                -
                -
String1         word    384  dup (?)
String2         word    384   dup (?)
```

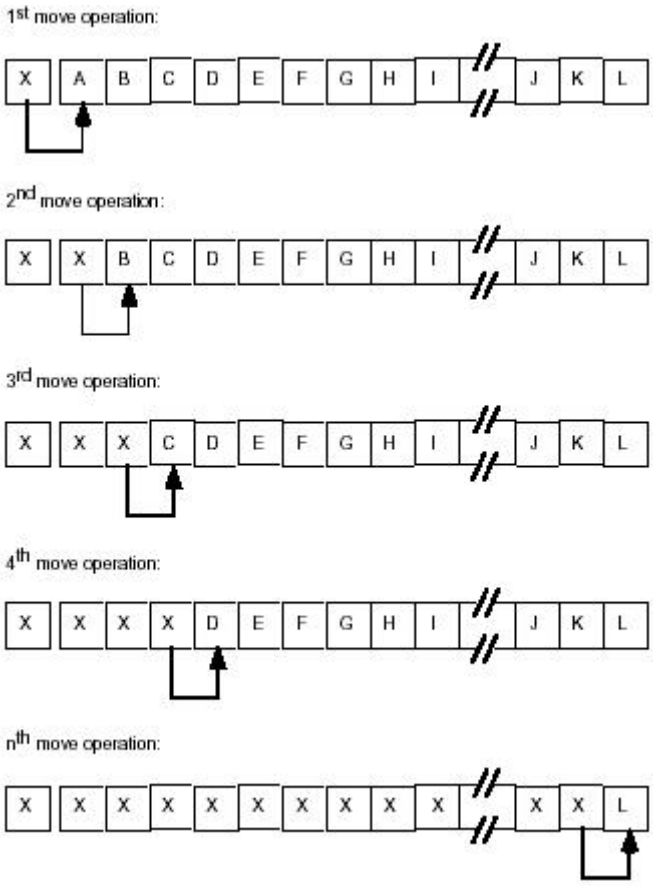
Pamiętamy, że cx zawiera element liczącym nie licznik bajtów. Kiedy używamy instrukcji movsw, CPU przesuwa liczbę słów określonych w rejestrze.

Jeśli ustawimy flagę kierunku przed wykonaniem instrukcji movsb / movsw / movsd, CPU zmniejszy rejestry si i di po przeniesieniu każdego elementu ciągu. To znaczy, że si i di muszą wskazywać koniec ich właściwych ciągów przed wywołaniem instrukcji movsb, movsw lub movsd. Na przykład

```
                std
                lea     si, String1+383
                lea     di,String2+383
                mov     cx, 384
        rep     movsb
                -
                -
                -
String1         byte    384  dup (?)
String2         byte    384   dup (?)
```

Chociaż są chwile kiedy przetwarzanie ciągu od końca do początku jest użyteczne (zobacz opis cmps w następnej sekcji), generalnie będziemy przetwarzać ciągi w kierunku do przodu ponieważ jest to dużo prostsze do zrobienia Jest jedna klasa operacji na ciągach gdzie przetwarzanie ciągów w obu kierunkach jest całkowicie obowiązkowe: przetwarzanie ciągów kiedy blok źródłowy i przeznaczenia zachodzą na siebie. Rozważmy co zdarzy się w następującym kodzie:

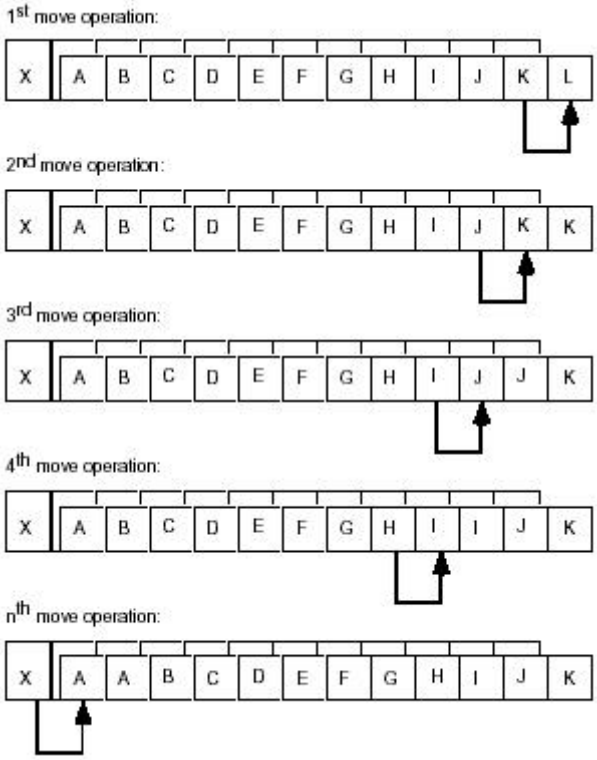
```
                cld
                lea     si, String1
                lea     di, String2
                mov     cx, 384
        rep     movsb
                -
                -
                -
String1         byte    ?
String2         byte    384  dup (?)
```



Rysunek 15.1 :Nadpisywanie danych podczas operacji przenoszenia bloków

Ta sekwencja instrukcji traktuje String1 i String2 jako parę 384 bajtowych ciągów. Jednak ostanie 383 bajty w tablicy String1 nachodzi na pierwsze 383 bajty w tablicy String2. Prześledźmy działanie tego kodu bajt po bajcie.

Kiedy CPU wykonuje instrukcje movsb, kopiuje bajt spod ds.:si (string1) do bajtu wskazywanego przez es:di (String2). Potem zwiększa si i di, zmniejsza cx o jeden i powtarza ten proces. Teraz rejestr si wskazuje na String1+1 (który jest adresem String2) a rejestr di wskazuje na String2+1. Instrukcja movsb kopiuje bajt wskazywany przez si do bajtu wskazywanego przez di. Jednak jest to bajt pierwotnie skopiowany z lokacji String1. Więc instrukcja movsb kopiuje pierwotną wartość z lokacji String1 do obu lokacji String2 i String2+1. Ponownie, CPU zwiększa si i di, zmniejsza cx i powtarza tą operację. Teraz instrukcja movsb kopiuje bajt z lokacji String1+2 (String2+1) do lokacji String2+2. Ale znowu, jest to wartość, która pierwotnie pojawiła się w lokacji String1. Każde powtórzenie pętli kopiuj następny element w String1 do następnej dostępnej lokacji w tablicy String2. Wygląda to tak jak na rysunku 15.1



Rysunek 15.2 : Poprawny sposób przenoszenia danych przy operacji przenoszenia bloku

Końcowy wynik jest taki, że X zostaje zreplikowany w całym ciągu. Instrukcja przeniesienia kopiuje argument źródłowy do komórki pamięci, która staje się operandem źródłowym dla następnych operacji przenoszenia, które powodują replikację.

Jeśli rzeczywiście chcemy przesunąć jedną tablicę do innej kiedy nachodzą n siebie, powinniśmy przenieść każdy element ciągu źródłowego do ciągu przeznaczenia poczynając od końca dwóch ciągów jak pokazano na rysunku 15.2

Ustawienie flagi kierunku i wskazywanie si i di na koniec ciągów pozwoli nam (poprawnie) przenieść jeden ciąg do innego, kiedy dwa ciągi zachodzą na siebie a ciąg źródłowy zaczyna się od mniejszego adresu niż ciąg przeznaczenia. Jeśli dwa ciągi nachodzą na siebie a ciąg źródłowy zaczyna się od wyższego adresu niż ciąg przeznaczenia, wtedy zerujemy flagę kierunku a si i di wskazują początek dwóch ciągów.

Jeśli dwa ciągi nie zachodzą na siebie, wtedy możemy łatwo użyć techniki przeniesienia ciągów w pamięci. Ogólnie działanie przy wyzerowanej fladze kierunku jest łatwiejsze.

Nie powinniśmy używać instrukcji movs do wypełniania tablicy wartością pojedynczego bajtu, słowa lub podwójnego słowa. Inna instrukcja ciągu, stos, jest dużo lepsza do tego celu. Jednakże, dla tablic, których elementy są większe niż cztery bajty, możemy zastosować instrukcję movs do inicjalizacji całej tablicy zawartością pierwszego elementu.

15.1.5 INSTRUKCJA CMPS

Instrukcja cmps porównuje dwa ciągi. CPU porównuje ciąg odnosząc się przez es:di do ciągu wskazywanego przez ds.:si. Cx zawiera długość dwóch ciągów (kiedy używamy przedrostka rep). Podobnie jak instrukcja movs, MASM pozwala na kilka różnych form tej instrukcji:

```
{REPE}      CMPSB
{REPE}      CMPSW
{REPE}      CMPSD                   ;dostępna tylko na 80386 i późniejszych
```

```
{REPE}      CMPS   przeznaczenie źródło
{REPNE}     CMPSB
{REPNE}     CMPSW
{REPNE}     CMPSD                          ;dostępna tylko na 80386 i późniejszych
{REPNE}     CMPS   przeznaczenie, źródło
```

Podobnie jak przy instrukcji movs, argumenty obecne w polu argumentów instrukcji cmps określają rozmiar argumentu. Określamy rzeczywisty adres argumentu w rejestrach si i di.

Bez przedrostka powtarzania, instrukcja cmps odejmuj wartość spod lokacji es:di od wartości spod ds.:si i uaktualnia flagi. Poza uaktualnieniem flag CPU nie używa różnicy stworzonej przez to odejmowanie. Po porównaniu dwóch komórek, cmps zwiększa lub zmniejsza rejestry si i di o jeden, dwa lub cztery (odpowiednio dla cmpsb / cmpsw / cmpsd). Cmps zwiększa rejestry si i di jeśli flaga kierunku jest wyzerowana i zmniejsza je w przeciwnym wypadku.

Oczywiście nie wykorzystamy rzeczywistej siły instrukcji cmps używając jej do porównywania pojedynczych bajtów lub słów w pamięci. Instrukcja ta bryluje kiedy używamy jej do całych ciągów. Z cmps, możemy porównywać kolejne elementy ciągu dopóki nie znajdziemy dopasowania lub dopóki nie dopasujemy kolejnych elementów.

Aby porównać dwa ciągi aby zobaczyć czy są równe lub nie równe, musimy porównywać odpowiadające sobie elementy w ciągu dopóki się nie skończą. Rozważmy następujące ciągi:

„String1”
„String1”

Jedyny sposób określenia czy te dwa ciągi są równe jest porównanie każdego znaku z pierwszego ciągu z odpowiadającym mu znakiem w drugim. W końcu drugi ciąg mógłby to być „String2” który zdecydowanie nie jest równy „String1”. Oczywiście, napotkawszy znak w ciągu przeznaczenia, który nie jest równy odpowiedniemu znakowi w ciągu źródłowym, porównanie jest zatrzymane. Nie potrzebujemy porównywać żadnego innego znaku w tych ciągach .

Przedrostek repe realizuje to zadanie. Będzie porównywał następujące po sobie elementy w ciągu tak długo jak są one równe a cx jest większe od zera. Możemy porównać dwa powyższe ciągi używając następującego kodu asemblerowego 80x86:

```
;Zakładamy, że ciągi są w tym samym segmencie a ES i DS. , ba wskazują na ten segment
                cld
                lea     si, AdrsSring1
                lea     di, AdrsString2
                mov     cx, 7
        repe    cmpsb
```

Po wykonaniu instrukcji cmpsb, możemy przetestować flagi używając standardowych instrukcji skoków warunkowych. Pozwoli to nam sprawdzenie równości , nierówności, mniejsze niż, większe niż itp.

Ciągi znaków są zazwyczaj porównywane przy zastosowaniu porządku leksykograficznego. W tym porządku najmniej znaczący element ciągu ma największą wagę. Jest to w bezpośrednim kontraście ze standardowym porównaniem całkowitym gdzie najbardziej znacząca część liczby niesie największą wagę. Ponadto, długość ciągu wpływa na porównanie tylko jeśli dwa ciągi są identyczne do długości krótszego ciągu. Na przykład, „Zebra” jest mniejsza niż „Zebras” ponieważ jest krótszym z dwóch ciągów, jednak „Zebra” jest większa niż „AAAAAAAAAAH!”, chociaż jest krótsza. Porównania leksykograficzne porównują odpowiednie elementy dopóki nie napotkają nie pasującego znaku lub końca krótszego ciągu. Jeśli para odpowiednich znaków nie pasuje, wtedy algorytm ten porównuje dwa ciągi opierając się n tym pojedynczym znaku. Jeśli te dwa ciągi pasują do długości krótszego ciągu, musimy porównać ich długości. Dwa ciągi są równe jeśli , i tylko jeśli, ich długości są równe i każda odpowiadająca sobie para znaków w tych ciągach jest identyczna. Porządek leksykograficzny jest standardowym porządkiem alfabetycznym z jakim dorastaliśmy.

Dla ciągów znakowych używamy instrukcji cmps w następujący sposób:

- Flaga kierunku musi być wyzerowana przed porównaniem ciągów
- Używamy instrukcji cmpsb do porównania ciągów na podstawie bajt przez bajt
  Nawet jeśli ciągi zawierają parzystą liczbę znaków, nie możemy użyć instrukcji cmpsw. Ona ni porównuje ciągów w porządku leksykograficznym.
- Rejestr cx musi być załadowany długością krótszego ciągu
- Używamy przedrostka repe
- Rejestry ds.:si i es:di muszą wskazywać absolutnie pierwszy znak w tych dwóch ciągach, które chcemy porównać

Po wykonaniu instrukcji cmps, jeśli dwa ciągi były równe, ich długości muszą być porównane żeby zakończyć porównanie. Poniższy kod porównuje parę ciągów znakowych:

```
                    lea     si, źródło
                    lea     di, przeznaczenie
                    mov     cx, lengthSource
                    mov     ax, lenghtDest
                    cmp     cx, ax
                    ja      Noswap
                    xchg    ax, cx
NoSwap: repe        cmpsb
                    jne     NotEqual
                    mov     ax, lengthSource
                    cmp     ax, lengthDest
NotEqual:
```

Jeśli używamy bajtów do przetrzymania długości ciągów, powinniśmy zmodyfikować stosownie ten kod.

Możemy również użyć instrukcji cmps do porównania wartość wielo słowa całkowitego (to jest wartość całkowita o podwyższonej precyzji). Z powodu ilości ustawień wymaganych dla porównania ciągów, nie jest to praktyczne dla wartości całkowitych mniejszych niż o długości trzech lub czterech słów, ale dla dużych wartości całkowitych jest to doskonały sposób porównania takich wartości. W odróżnieniu od ciągów znaków, nie możemy porównać ciągów całkowitych przy użyciu porządku leksykograficznego. Kiedy porównujemy ciągi, porównujemy znaki od najmniej znaczącego bajtu do najbardziej znaczącego bajtu. Kiedy porównujmy wartości całkowite musimy porównać wartości od najbardziej znaczącego bajtu (lub słowa / podwójnego słowa) w dół do najmniej znaczącego bajtu, słowa lub podwójnego słowa. Więc dla porównania dwóch 128 bitowych wartości całkowitych użyjemy następującego kodu na 80286:

```
                    std
                    lea     si, SourceInteger+14
                    lea     di, estInteger+14
                    mov     cx, 8
            repe    cmpsw
```

Kod ten porównuje wartości całkowite od ich najbardziej znaczących słów w dół do najmniej znaczącego słowa. Instrukcja cmpsw kończy się kiedy dwie wartości są nie równe lub przy zmniejszeniu cx do zera (zakładając ,że dwie wartości są równe). Ponownie flagi uwzględniają wynik porównania .

Przedrostek repne instruuje instrukcję cmps aby porównywała kolejne elementy ciągu tak długo dopóki nie będą pasować. Flagi 80x86 są używane po wykonaniu tej instrukcji. Albo rejestr cx zawiera zero (w takim przypadku dwa ciągi są całkowicie różne lub zawiera liczbę elementów porównywanych aż do dopasowania. Ta forma instrukcji cmps nie jest szczególnie użyteczna przy porównywaniu ciągów, jest użyteczna do zlokalizowania pierwszej pary dopasowanych pozycji w parz tablic bajtowych lub słów. Ogólnie jednak będziemy rzadko używali przedrostka repne z cmps.

Ostatnią rzeczą do zapamiętania przy stosowaniu instrukcji cmps – wartość w rejestrze cx określa liczbę elementów do przetworzenia a nie liczbę bajtów. Dlatego też, kiedy używamy cmpsw, cx określa liczbę słów do porównania. To oczywiście jest dwukrotna liczba bajtów do porównania.

15.1.6 INSTRUKCA SCAS

Instrukcja cmps porównuje dwa ciągi ze sobą. Nie możemy użyć jej do wyszukania określonego elementu wewnątrz ciągu. Na przykład nie możemy zastosować instrukcji cmps do szybkiego wyszukania zera w jakimś innym ciągu. Możemy użyć instrukcji scas (przeszukaj ciąg) dla tego zadania.

W odróżnieniu od instrukcji movs i cmps, instrukcja scas wymaga tylko ciągu przeznaczenia (es:di) zamiast obu ciągów źródłowego i przeznaczenia. Operand źródłowy jest wartością w rejestrze al. (scasb), ax (scasw) lub eax (scasd) .Instrukcja scas, porównuje wartość z akumulatora (al., ax lub eax) z wartością wskazywaną przez es:di a potem zwiększa (lub zmniejsza) di o jeden, dwa lub cztery. CPU ustawia flagi stosownie do wyniku porównania. Chociaż to może być użyteczna czasami, scas jest dużo bardziej użyteczna kiedy stosujmy przedrostki repe i repne

Kiedy jest obecny przedrostek repe (powtarzaj kiedy równe), scas przeszukuje ciąg poszukując elementu, który nie pasuje do wartości w akumulatorze. Kiedy użyjemy przedrostka repne (powtarzaj kiedy nie równe) ,scas przeszukuje ciąg poszukując pierwszego elementu ciągu, który jest równy wartości w akumulatorze.

Możemy być zdumieni „dlaczego te przedrostki robią dokładnie co innego niż powinny robić?”. Paragraf powyżej nie całkiem poprawnie wyraził działanie instrukcji scas. Kiedy używamy przedrostka repe ze scas, 80x86 przeszukuje cały ciąg dopóki wartość w akumulatorze jest równa argumentowi ciągu. Jest to odpowiednik przeszukiwania całego ciągu do pierwszego elementu, który nie jest dopasowany do wartości w akumulatorze. Instrukcja scas z repne przeszukuje cały ciąg dopóki akumulator nie jest równy argumentowi ciągu. Oczywiście ta postać poszukuje pierwszej wartości w ciągu dopasowanej do wartości w rejestrze akumulatora. Instrukcja scas przybiera następujące formy:

```
{REPE}      SCASB
{REPE}      SCASW
{REPE}      SCASD           ;dostępny tylko na 80386 i późniejszych procesorach
{REPE}      SCAS        przeznaczenie
{REPNE}     SCASB
{REPNE}     SCASW
{REPNE}     SCASD           ;dostępny tylko na 80386 i późniejszych procesorach
{REPNE}     SCAS        przeznaczenie
```

Podobnie jak przy instrukcjach cmps i movs., wartość w rejestrze cx określa liczę elementów do przetworzenia, nie bajtów, kiedy stosujemy przedrostek powtórzenia.

15.1.7 INSTRUKCA STOS

Instrukcja stos przechowuje wartość z akumulatora pod lokacją określoną przez es:di. Po przechowaniu tej wartości, CPU zwiększa lub zmniejsza di w zależności od stanu flagi kierunku. Chociaż instrukcja stos ma wiele zastosowań, podstawowym zastosowaniem jest inicjalizowanie tablic i ciągów stałymi wartościami. Na przykład jeśli mamy 265 bajtową tablicę i chcemy wypełnić ją zerami , użyjemy następującego kodu:
;Przypuszczalnie rejestr ES już wskazuje segment zawierając DestString

```
        cld
        lea     di, DestString
        mov     cx, 128                 ;256 bajtów to 128 słów
        xor     ax, ax                  ; AX := 0
rep     stosw
```

Kod ten zapisuje 128 słów zamiast 256 bajtów ponieważ pojedyncza operacja stosw jest szybsza niż dwie operacje stosb. Na 80386 lub późniejszych kod ten może zapisać 64 podwójnych słów wykonując to samo szybciej.

Instrukcja stos przybiera cztery formy. Oto one:

```
{REP}   STOSB
{REP}   STOSW
{REP}   STOSD
{REP}   STOS    przeznaczenie
```

Instrukcja stosb przechowuje wartość z rejestru al. W określonej komórce(-ach) pamięci, instrukcja stosw przechowuje rejestr ax w określonej komórce –ach) pamięci a instrukcja stosd przechowuje eax w określonej lokacji. Instrukcja stos jest instrukcją albo instrukcją stosb, stosw lub stosd w zależności od rozmiaru określonego argumentu.

Zapamiętajmy, że instrukcja stos jest użyteczna tylko dla inicjalizowania tablic bajtu, słowa lub podwójnego słowa stałymi wartościami. Jeśli musimy zainicjalizować tablicę różnymi wartościami, nie możemy użyć instrukcji stos. Możemy użyć movs w takiej sytuacji.

15.1.8 INSTRUKCJA LODS

Instrukcja lods jest unikalna pośród instrukcji ciągów. Nigdy nie będziemy używali przedrostka powtórzenia z tą instrukcją. Instrukcja lods kopiuje bajt lub słowo wskazywane przez ds.:si do rejestru al., ax lub eax, poczym zwiększa lub zmniejsza rejestr si o jeden , dwa lub cztery. Powtarzanie tej instrukcji poprzez przedrostek powtórzenia nie służyć będzie jakimkolwiek celom ponieważ rejestr akumulatora będzie nadpisywany za każdym razem, kiedy będzie powtarzana instrukcja lods. Na koniec operacji powtarzania akumulator będzie zawierał ostatnią wartość odczytaną z pamięci.

Zamiast tego używamy instrukcję lods do pobierania bajtów (lodsb) , słów (lodsw) lub podwójnych słów (lodsd) z pamięci do dalszego przetwarzania. Przez użycie instrukcji stos możemy zsyntetyzować silniejsze operacje na ciągach.

Podobnie jak instrukcja stos, instrukcja lods przybiera cztery formy:

```
{REP}  LODSB
{REP}  LODSW
{REP}  LODSD                ;dostępna na 80386 lub późniejszych
{REP}  LODS  przeznaczenie
```

Jak wspomniano wcześniej, będziemy rzadko, jeśli w ogóle, używali przedrostka rep z tymi instrukcjami. 80x86 zwiększa lub zmniejsza si o jeden, dwa lub cztery w zależności od flagi kierunku i czy użyliśmy instrukcji lodsb, lodsw lub lodsd.

15.1.9 BUDOWANIE ZŁOŻONYCH FUNKCJI CIĄGÓW Z LODS I STOS

80x86 utrzymuje tylko pięć różnych instrukcji na ciągach: movs, cmps, scas, lods i stos. Nie są to z pewnością jedyne operacje na ciągach jakie chcielibyśmy stosować. Jednak możemy użyć instrukcji lods i stos do łatwego wygenerowania jakiejś szczególnej operacji na ciągach. Na przykład przypuśćmy, że chcemy operacji na ciągach, która konwertuje wszystkie duże znaki w ciągu na małe. Możemy użyć następującego kodu:

```
;Przypuszczalnie ES i DS. zostały ustawione aby wskazywały ten sam segment zawierający ciąg do konwersji

                lea     si, String2Convert
                mov     di, si
                mov     cx, LenghtOfString
Convert2Lower:  lodsb                       ;pobranie następnego znaku z ciągu
                cmp     al., 'A'            ;czy duży znak?
                jb      NotUpper
                cmp     al., 'Z'
                ja      NotUpper
                or      al., 20h            ;konwersja na małą literę
NotUpper        stosb                       ;przechowanie w przeznaczeniu
                loop    Convert2Lower
```

Zakładając, że chcemy zmarnować 256 bajtów na tablicę, ta konwersja może być przyśpieszona przez użycie instrukcji xlat:

```
;Przypuszczalnie, ES i DS. zostały ustawione tak, aby wskazywały ten sam segment, zawierający ciąg do
;konwersji

                cld
                lea     si, String2Convert
                mov     di, si
                mov     cx, LengthOfString
                lea     bx, ConversionTable
Convert2Lower:  lodsb                       ;pobranie następnego znaku w ciągu
                xlat                        ;właściwa konwersja
                stosb                       ;przechowanie w przeznaczeniu
                loop    Convert2Lower
```

Tablica konwersji oczywiście będzie zawierała indeks do tablicy do każdej lokacji z wyjątkiem offsetów 41h....5Ah. Pod tymi lokacjami tablica konwersji będzie zawierała wartości 61h....7Ah (tj. indeksy 'A'...'Z' tablica zawierałaby kody dla 'a'...'z')

Ponieważ instrukcje lods i stos używają akumulatora jako pośrednika ,możemy użyć każdej operacji akumulatora do szybkiego manipulowania elementami ciągu.

15.1.10 PRZEDROSTKI I INSTRUCKJE CIĄGÓW

Instrukcje ciągów akceptują przedrostki segmentów, przedrostki blokady i przedrostki powtórzenia. Faktycznie można określić wszystkie trzy typy przedrostków instrukcji które są pożądane. Jednakże, wskutek tego wystąpią błędy we wcześniejszych chipach 80x86 (przed 80386), wic nie powinniśmy używać więcej niż pojedynczego przedrostka (powtórzenia, blokady lub przesłonięcia segmentu) w instrukcjach ciągów, chyba ,że nasz kod będzie działał na późniejszych procesorach; prawdopodobnie nawet w obecnych dniach. Jeśli

koniecznie musimy użyć dwóch lub więcej przedrostków i uruchomić na wcześniejszych procesorach, upewnijmy się , ze wyłączyliśmy przerwania podczas wykonywania instrukcji na ciągach.

15.2 CIĄGI ZNKÓW

Ponieważ będziemy natykali się na ciągi znaków częściej niż inne typy ciągów, zasługują one na specjalną uwagę. Poniższa sekcja opisuje ciągi znaków i różne typy operacji na ciągach.

15.2.1 TYPY CIĄGÓW

Na większości podstawowych poziomów, instrukcje ciągów 80x86 działają tylko na tablicach znaków. Jednakże, ponieważ wiele typów danych ciągów zawiera tablicę znaków jako składnik, instrukcje ciągów są przydatne do manipulowania częścią ciągu.

Prawdopodobnie największą różnicą pomiędzy ciągiem znaków a tablicą znaków jest atrybut długości. Tablica znaków zawiera stałą liczbę znaków. Ani mniej ani więcej. Ciąg znaków ma dynamiczną długość podczas wykonania, to znaczy, liczba znaków zawartych w ciągu w programie. Ciągi znaków w odróżnieniu od tablic znaków, mają zdolność do zmiany swojego rozmiaru podczas wykonywania (wewnątrz pewnego zakresu oczywiście).

Komplikuje sprawy to, że są dwa ogólne typy ciągów: ciągi alokowane statycznie i ciągi alokowane dynamicznie. Ciągi alokowane statycznie są dane jako stałe, o maksymalnej długości tworzonej w czasie programu. Długość ciągu może różnić się w czasie wykonania ale tylko między zero a tą maksymalną długością. Większość systemów alokuje i dealokuje ciągi alokowane dynamicznie w obszarze pamięci kiedy używamy ciągów. Takie ciągi mogą być dowolnej długości (do jakiejś sensownej maksymalnej granicy). Dostęp do takich ciągów jest mniej wydajny niż dostęp do ciągów alokowanych statycznie. Ponadto odzyskiwanie pamięci może zabrać trochę czasu. Pomimo to, ciągi alokowane dynamicznie są dużo bardziej wydajne przestrzennie niż ciągi alokowane statycznie i , w tym przypadku, dostęp do ciągów alokowanych dynamicznie jest również szybszy. Większość z przykładów tego rozdziału będzie stosowało ciągi alokowane statycznie.

Ciągi o dynamicznej długości potrzebują jakiegoś sposobu na śledzenie tej długości. Podczas gdy jest kilka możliwych sposobów przedstawiania długości ciągów, da najbardziej popularne to ciąg z przedrostkiem długości i ciąg zakończony zerem. Ciąg z przedrostkiem długości skała się z pojedynczego bajtu lub słowa, które zawiera długość ciągu. Bezpośrednio po wartości długości są znaki tworzące ciąg. Zakładając użycie bajtowego przedrostka długości możemy zdefiniować ciąg „HELLO” jak następuje:

```
HelloStr        byte    5, „HELLO”
```

Ciągi z przedrostkiem długości są nazywane często ciągami Pascalowymi ponieważ jest to typ zmiennej ciągu wspierana przez większość wersji Pascala.

Innym popularnym sposobem określania długości ciągu jest użycie ciągu zakończonego zerem. Ciąg zakończony zerem składa się z ciągu znaków zakończonych bajtem zerowym. Ten typ ciągów jest często nazywany ciągami C ponieważ są one typem używanym przez C/C++. Ponieważ Standardowa Biblioteka UCR naśladuje standardową bibliotekę C, również ciągów zakończonych zerem.

Ciągi Pascalowe są dużo lepsze niż ciągi C/C++ z kilku powodów. Po pierwsze , obliczenie długości ciągu pascalowego jest banalne. Musimy pobrać tylko pierwszy bajt (słowo) ciągu i możemy obliczyć jego długość. Obliczanie długości ciągu C/C++ jest zdecydowanie mniej wydajne. Musimy przeszukać cały ciąg (np. używając instrukcji scasb) aż do bajtu zero. Jeśli ciąg C/C++ jest długi, może to zająć jakiś czas. Co więcej ciąg C/C++ nie może zawierać znaku NULL Z drugiej strony , ciągi C/C++ mogą być każdej długości, ale wymagają obciążającego dodatkowego pojedynczego bajtu. Ciągi Pascalowe jednak nie mogą być dłuższe niż 255 znaków kiedy używamy tylko długości pojedynczego bajtu. Dla ciągów dłuższych niż 255 bajtów, będziemy potrzebowali dwóch bajtów do przechowania długości ciągu Pascalowego. Ponieważ większość ciągów jest mniejsza niż 256 znaków długości, to nie jest wielką wadą.

Zaletą ciągów zakończonych zerem jest to ,że są łatwe do użycia w programach języka asemblera. Jest to szczególnie prawdziwe dla ciągów, które są tak długie, że wymagają wielu lini kodu źródłowego w programach asemblerowych. Zliczanie każdego znaku w ciągu jest tak nużące, ze nie jest warte. Jednakże możemy napisać makro, które łatwo zbuduj dla nas ciąg Pascalowski:

```
Pstring         macro   String
                local   StringLength, StringStart
                byte    StringLength
StringStart     byte    String
StringLength    =       $ - StringStart
                endm
```

```
-
-
-
Pstring   „Ten ciąg ma przedrostek długości”
```

Tak długo jak ciąg mieści się całkowicie w jednej lini źródłowej, możemy zastosować to makro do generowania ciągów w stylu Pascalowskim.

Popularne funkcje ciągów. Takie jak konkatenacji, długości, odejmowani, indeks i inne są dużo łatwiejsze do napisania kiedy używamy ciągów z przedrostkiem długości. Więc będziemy używali ciągów Pascalowskich. Co więcej, Standardowa Biblioteka UCR dostarcza dużej liczby funkcji ciągów C/C++, wic nie ma potrzeby powielania tych funkcji tutaj.

15.2.2 PRZYPISYWANIE CIĄGÓW

Możemy łatwo przypisać jeden ciąg do innego przy użyciu instrukcji movsb. Na przykład, jeśli chcemy przypisać ciąg z przedrostkiem długości String1 do String2 użyjemy czegoś takiego:

```
;Przypuszczalnie ES i DS są już ustawione

                lea     si, Sring1
                lea     di, String2
                mov     ch, 0                   ;rozszerzamy len do 16 bitów
                mov     cl, String1             ;pobieranie długości ciągu
                inc     cx                      ;obejmujemy długość bajtu
rep             movsb
```

Kod ten zwiększa cx o jeden przed wykonaniem movsb ponieważ długość bajtu zawiera długość ciągu zawierającą samą długość bajtu.

Ogólnie zmienna napisowa może być zainicjalizowana stałymi przez użycie makra Pstring opisanego wcześniej. Jednak, jeśli musimy ustawić zmienną napisową na jakąś wartość stałą, możemy napisać podprogram StrAssign, który przypisuje ciąg bezpośrednio po call. Poniższa procedura właśnie to robi :

```
                include         stdlib.a
                includelib      stdlib.lib

cseg            segment         para public ‘code’
                assume          cs: cseg, ds.: dseg, es: dseg, ss: sseg

;procedura przypisania ciągu

MainPgm         proc    far
                mov     ax , seg  dseg
                mov     ds., ax
                mov     es, ax

                lea     di, ToString
                call    StrAssign
                byte    „To jest przykład jak „
                byte    „jest używany podprogram StrAssign”
                nop
                ExitPgm
MainPgm

StrAssign       proc    near
                push    bp
                mov     bp, sp
                pushf
                push    ds.
                push    si
                push    di
```

```
                        push    cx
                        push    ax
                        push    di                      ;zachowanie do późniejszego użycia
                        push    es
                        cld
;Pobranie adresu ciągu źródłowego
                        mov     ax, cx
                        mov     es, ax
                        mov     di, 2[bp]               ;pobranie adresu powrotu
                        mov     cx, 0ffffh              ;szukanie tak długo jak pobiera
                        mov     al., 0                  ;szukanie zera
                repne   scasb                           ;obliczanie długości ciągu
                        neg     cx                      ;konwersja długości do liczby dodatniej
                        dec     cx                      ;ponieważ zaczynamy od –1 nie 0
                        dec     cx                      ;przeskakujemy bajt zakończony zerem
;Teraz kopiujemy ciągi
                        pop     es                      ;pobranie segmentu przeznaczenia
                        pop     di                      ;pobranie adresu przeznaczenia
                        mov     al., cl                 ;przechowanie bajtu długości
                        stosb
;Teraz kopiujemy ciąg źródłowy
                        mov     ax, cs
                        mov     ds ,ax
                        mov     si, 2[bp]
                rep     movsb

;Uaktualniamy adres powrotny i zostawiamy:
                        inc     si                      ;przeskakujemy bajt zerowy
                        mov     2[bp], si

                        pop     ax
                        pop     cx
                        pop     di
                        pop     si
                        pop     ds.
                        popf
                        pop     bp
                        ret
StrAssign               endp

cseg                    ends

dseg                    segment   para public  'data'
ToString                byte      255 dup  (0)
dseg                    ends

sseg                    segment  para  stack  'stack'
                        word      256  dup (?)
sseg                    ends
                        end       MainPgm
```

Kod ten używa instrukcji scas do określenia długości ciągu bezpośrednio następującego po instrukcji call. Ponieważ kod określa długość, przechowuje tą długość w pierwszym bajcie ciągu przeznaczenia a potem kopiuje tekst następujący po call do zmiennej napisowej. Po skopiowaniu ciągu, kod ten modyfikuje adres powrotny aby wskazywał poza bajt zakończony zerem. Potem procedura przekazuje sterowanie do kodu wywołującego.

Oczywiście ta procedura przypisywania ciągów nie jest bardzo wydajna, ale jest bardzo łatwa do użycia. Ustawienie es:di jest takie, że wszystko co trzeba zrobić to użyć tej procedury. Jeśli potrzebujemy szybszego przypisywania ciągów po prostu użyjemy instrukcji movs:

```
;Przypuszczalnie, DS. i ES już były ustawione
```

```
                    lea     si, SourceString
                    lea     di, DestString
                    mov     cx, LengthSource
              rep   movsb
                    -
                    -
                    -
SourceString        byte    LengthSource –1
                    byte    „To jest przykład jak „
                    byte    „ jest używany podprogram StrAssign „
LengthSource        =       $ - SourceString

DestString          byte    256  dup (?)
```

Zastosowanie instrukcji wbudowanych wymaga znacznie więcej ustawień (i pisania!), ale jest dużo szybsze niż procedura StrAssign. Jeśli nie lubisz pisania, możesz zawsze napisać makro, które przypisze ciąg za ciebie.

15.2.3 PORÓWNANIE CIĄGÓW

Porównywanie dwóch ciągów znakowych było już omawiane w sekcji o instrukcji cmps. Dostarczymy konkretnego przykładu, tak, że nie będzie powodu aby rozpatrywać tego tematu później.

Notka: wszystkie poniższe przykłady zakładają, że es i ds wskazują właściwe segmenty zawierające ciągi przeznaczenia i źródłowy.

Porównanie Str1 z Str2:

```
                    lea     si, Str1
                    lea     di, Str2

;pobranie minimalnej długości dwóch ciągów
                    mov     al., Str1
                    mov     cl, Str2
                    cmp     al., Str2
                    jb      CmpStrs
                    mov     cl, Str2
;porównanie dwóch ciągów

CmpStrs”            mov     ch, 0
                    cld
              repe  cmpsb
                    jne     StrsNotEqual

;Jeśli CMPS stwierdzi, że są równe, porównuje ich długości, dla pewności
                    cmp     al., Str2
StrsNotEqual:
```

Przy etykiecie StrsNotEqual, flagi będą zawierały wszystkie adekwatne informacje o miejscach tych dwóch ciągów, Możemy użyć instrukcji skoków warunkowych do przetestowania wyników tego porównania.

15.3 FUNKCJE CIĄGÓW ZNKÓW

Większość jeżyków wysokiego poziomu takie jak Pascal, BASIC, C i PL/I dostarcza kilku funkcji i procedur (albo wbudowanych w język albo jako część biblioteki standardowej). Inaczej niż przy pięciu ,powyższych ,operacjach na ciągach , 80x86 nie wspiera żadnej funkcji na ciągach. Dlatego też jeśli potrzebujemy określonej funkcji ciągu, będziemy musieli napisać ją sami. Poniższe sekcje opisują wiele z popularnych funkcji ciągów i jak zaimplementować je w asemblerze.

15.3.1 SUBSTR

Funkcja Substr kopiuje część jednego ciągu do innego. W języku wysokopoziomowym funkcja ta zazwyczaj przybiera postać:

DestStr := Substr (SrcStr, Index, Length);

gdzie:

- DestStr jest nazwą zmiennej napisowej gdzie chcemy przechować pod- ciąg
- SrcStr jest nazwą ciągu źródłowego (z którego jest pobierany podciąg)
- Index jest pozycją startową znaku wewnątrz ciągu (1..length(SrcStr)), i
- Length jest to długość pod-ciągu jaki chcemy skopiować do DestStr

Poniższy przykład pokazuje jak działa Substr.

```
SrcStr := 'This is an example of a string';
DestStr := Substr (SrcStr, 11,7);
write(DestStr);
```

Wydrukuje to „example". Wartość indeksu to jedenaście, więc funkcja Substr zacznie kopiowanie danych poczynając od jedenastego znaku w ciągu. Jedenasty znak to „e" w „example". Długość tego ciągu to siedem.

To wywołanie kopiuje siedem znaków „example" do DestStr

```
SrcStr := 'This is an example of a string';
DestStr := Substr (SrcStr, 1,10);
write(DestStr);
```

To wydrukuje „This is an". Ponieważ indeks to jeden, to wystąpienie funkcji Substr zacznie kopiowanie 10 znaków poczynając od pierwszego znaku w ciągu.

```
SrcStr := 'This is an example of a string';
DestStr := Substr (SrcStr, 20,11);
write(DestStr);
```

To wydrukuje 'of a string'. To wywołanie Substr wydobędzie ostatnie jedenaście znaków z ciągu.

Co zdarzy się jeśli indeks i wartość długości znajdą się poza zakresem? Na przykład, co zdarzy się jeśli index będzie zerem lub większy niż długość ciągu? Co zdarzy się jeśli index jest OK., ale suma index i length jest większa niż długość ciągu źródłowego? Możemy zadziałać w tych nienormalnych sytuacjach na jeden z trzech sposobów: (1) zignorować możliwość błędu ; (2) przerwać program z błędem wykonania; (3) przetworzyć jakąś sensowna liczbę znaków w odpowiedzi na to zgłoszenie.

Pierwsze rozwiązanie działa przy założeniu, że program wywołujący nigdy nie popełni pomyłki przy obliczaniu wartości dla parametrów funkcji Substr. Ślepo zakłada, że wartości przekazywane do funkcji Substr są poprawne i przetwarza ciąg w oparciu o te założenia. Może to dać dziwaczny efekt. Rozważmy następujący przykład, który używa ciągu z przedrostkiem długości:

```
SourceStr := '1234567890ABCDEFGHJIKLMNOPQRSTUVWXYZ';
DestStr := Substr (SrcStr, 0,5);
write('DestStr');
```

wydrukuje '$1234'. Powodem jest oczywiście to, że SourceStr jest ciągiem z przedrostkiem długości. Dlatego też długość 36 pojawi się pod offsetem zero wewnątrz ciągu. Jeśli Substr używa nielegalnego indeksu zero, wtedy długość ciągu będzie zwrócona jako pierwszy znak. W tym szczególnym przypadku, długość ciągu ,36, odpowiada kodowi znaku ASCII '$'.

Sytuacja jest znacznie gorsza jeśli wartość określona dla index jest ujemna lub większa niż długość ciągu. W takim przypadku funkcja Substr zwracałaby podciąg zawierający znaki pojawiające się przed lub po ciągu źródłowym. Nie jest to wynik odpowiedni.

Pomimo ignorowani problemów możliwych błędów w funkcji Substr, jest jedna duża zaleta przetwarzania podciągów w ten sposób: kod wynikowy Substr jest bardziej wydajny jeśli nie musimy wykonywać sprawdzania danych w czasie wykonywania. Jeśli wiemy, że wartości index i length są zawsze wewnątrz akceptowalnego zakresu, wtedy nie musimy wykonywać sprawdzania wewnątrz funkcji Substr. Jeśli możemy zagwarantować, ze błąd ni wystąpi, nasze programy będą działały (w pewnym stopniu)szybciej poprze wyeliminowania kontroli podczas wykonania.

Ponieważ większość programów rzadko jest wolnych od błędów, mówimy o dużym ryzyku zakładając, ze wszystkie wywołania podprogramu Substr przekazują odpowiednie wartości. Dlatego też pewna porcja kontroli w czasie wykonania jest często konieczna do wyłapania błędów w programie. Błędy wystąpią przy następujących warunkach:

- Parametr indeksowy (index) jest mniejszy niż jeden
- Index jest większy niż długość ciągu
- Parametr długości Substr (length) jest większy niż długość ciągu
- Suma index i length jest większa niż długość ciągu

Alternatywa dla ignorowania każdego z tych błędów jest przerwanie informacją o błędzie. Jest to prawdopodobnie dobre podczas fazy rozwoju programu, ale jeśli nasz program jest już w rękach użytkowników, może to być rzeczywista klęska. Nasi klienci nie byliby zbytnio szczęśliwi gdyby musieli spędzać wiele dni wprowadzając dane do programu, a on przerwałby działanie powodując zgubienie danych które wprowadzali. Alternatywą dla przerwania jeśli wystąpi błąd, jest posiadanie funkcji Substr zwracającej warunek błędu. Wtedy pozostawiamy to kodowi wywołującemu, określenie czy wystąpił błąd. Technika ta dział dobrze z trzecią alternatywą obsługiwania błędów: przetwarzaniem podciągów jak najlepiej można.

Trzecia alternatywa obsługi błędów jak najlepiej można jest prawdopodobnie najlepszą alternatywą. Obsługuje warunki błędu w następujący sposób:

- Parametr indeksowy (index) jest mniejszy niż. Są dwa sposoby operowania tym warunkiem błędu. Pierwszy sposób to automatyczne ustawienie parametru index na jeden i powrót do podciągu zaczynając od pierwszego znaku ciągu źródłowego. Inną alternatywą jest zwrot pustego ciągu, długość ciągu zero jako podciąg. Są również możliwe wariacje na ten temat. Możemy zwrócić podciąg zaczynając od pierwszego znaku jeśli index to zero a ciąg pusty jeśli index jest ujemny. Jeszcze inną alternatywą jest użycie liczb bez znaku. Wtedy musimy się martwić o przypadek kiedy index jest równy zero. Liczba ujemna, powinna w kodzie wywołującym przypadkiem wygenerować jeden, wyglądała by jak duża liczba dodatnia.
- Index jest większy niż długość ciągu .Jeśli jest taki przypadek, wtedy funkcja Substr powinna zwrócić pusty ciąg. Intuicyjnie jest to właściwa odpowiedź na tą sytuację.
- Parametr długości (Length) Substr jest większy niż długość ciągu - lub
- Suma Index i Length jest większa niż długość ciągu. Punkty trzeci czwarty maja taki sam problem, długość żądanego podciągu wychodzi poza koniec ciągu źródłowego. W tym przypadku Substr powinna zwracać podciąg składający się ze znaków zaczynających się pod Index aż do końca ciągu źródłowego.

Poniższy kod dla funkcji Substr oczekuje czterech parametrów: adresów ciągów źródłowego i przeznaczenia, indeksu startowego i długości żądanego podciągu. Substr oczekuje tych parametrów w następujących rejestrach:

ds:si	Adres ciągu źródłowego
es:di	Adres ciągu przeznaczenia
ch	Indeks startowy
cl	Długość podciągu

Substr zwraca następujące wartości:

- Podciąg pod lokacją es:di
- Substr zeruje flagę przeniesienia jeśli nie było błędów. Ustawia flagę przeniesienia jeśli był błąd
- Zachowuje wszystkie rejestry

Jeśli wystąpi błąd, wtedy kod wywołujący musi zanalizować wartości w si, di i cx aby określić dokładny powód powstania błędu (jeśli jest to konieczne). W przypadku błędu, funkcja Substr zwróci następujące podciągi:

- Jeśli parametr index (ch) to zero, Substr użyje jeden
- Oba parametry Index i Length są wartościami bajtowymi bez znaku, dlatego też nigdy nie są ujemne
- Jeśli parametr index jest większy niż długość ciągu źródłowego Substr zwróci pusty ciąg
- Jeśli suma parametrów Index i Length jest większa niż długość ciągu źródłowego, Substr zwróci tylko znaki od Index do końca ciągu źródłowego. Poniższy kod realizuje funkcję Podciągów

```
;funkcja Podciąg
;
;postać HLL
;
;procedure substring (var Src :string;
;                     Index, Length: integer;
;                     var Dest : string;
;
;Src – Adres ciągu źródłowego
```

```
;Index – Indeks do ciągu  źródłowego
;Length – długość podciągu do  wyekstrahowania
;Dest – adres ciągu przeznaczenia
;
;Kopiuje ciąg źródłowy  z adresu [Src+index] długości
;Length do ciągu przeznaczenia
;
;Jeśli wystąpi błąd, flaga przeniesienia jest zwracana jako ustawiona, w przeciwnym razie jako wyzerowana
;
;Parametry są przekazywane jak następuje:
;
;DS:SI – Adres ciągu źródłowego
;ES:DI – Adres ciągu przeznaczenia
;CH – Indeks do ciągu źródłowego
;CL – długość ciągu źródłowego
;
;Notka: ciągi wskazywane przez rejestry SI i DI są ciągami z przedrostkiem długości. To znaczy, pierwszy bajt
;każdego ciągu zawiera długość tego ciągu.

Substring       proc    near
                push    ax
                push    cx
                push    di
                push    si
                clc                             ;zakładamy brak błędu
                pushf                           ;zachowujemy stan flagi kierunku
;Sprawdzamy poprawność parametrów
                cmp     ch, [si]                ;czy index jest poza długością ciągu źródłowego?
                ja      ReturnEmpty
                mov     al., ch                 ;zobacz czy suma index i length jest poza końcem
                dec     al.                     ;ciągu
                add     al., cl
                jc      TooLong                 ;błąd jeśli > 255
                cmp     al., [si]               ;poza długością źródła?
                jbe     OkaySoFar

;Jeśli pociąg nie jest całkowicie zawarty wewnątrz ciągu źródłowego, obcina go:

TooLong:        popf
                stc                             ;zwrot flagi błędu
                pushf
                mov     al., [si]               ;pobranie maksymalnej długości
                sub     al., ch                 ;odjęcie wartości index
                inc     al.                     ;modyfikacja jeśli właściwe
                mov     cl, al.                 ;zachowanie nowej długości
OkaySoFar:      mov     es:[di], cl             ;zachowanie długości ciągu przeznaczenia
                inc     di
                mov     al., ch                 ;pobranie indeksu do źródła
                mov     ch, 0                   ;wartość długości poszerzonej zerem do CX
                mov     ah, 0                   ;indeks rozszerzony zerem do AX
                add     si, ax                  ;obliczanie adresu podciągu
                cld
        rep     movsb                           ;kopiowanie podciągu

                popf
SubStrDone:     pop     si
                pop     di
                pop     cx
                pop     ax
                ret
```

```
;Zwracanie pustego ciągu:

ReturnEmpty:    mov     byte  ptr  es:[di], 0
                popf
                stc
                jmp     SubStrDone
SubStrDone:     endp
```

15.3.2 INDEX

Funkcja ciągu index przeszukuje pierwszego wystąpienia jednego ciągu wewnątrz innego i zwraca offset do tego wystąpienia. Rozważmy następującą postać HLL'a:

```
SourceStr := 'Hello world';
TestStr : = 'world';
I := INDEX(SourceStr, TestStr);
```

Funkcja index szuka w całym ciągu źródłowym pierwszego wystąpienia ciągu testowego. Jeśli znajdzie, zwraca indeks do ciągu źródłowego gdzie zaczyna się ciąg testowy. W powyższym przykładzie funkcja INDEX zwróci siedem ponieważ podciąg 'world' zaczyna się od siódmej pozycji znaku w ciągu źródłowym.

Jedyny możliwy błąd wystąpi jeśli Index nie znajdzie ciągu testowego w ciągu źródłowym. W takiej sytuacji większość implementacji zwróci zero Nasza wersja będzie robiła podobnie. Funkcja Index wykonuje następujące działania w następujący sposób:

1) Porównuje długość ciągu testowego z długością ciągu źródłowego. Jeśli ciąg testowy jest dłuższy, Index natychmiast zwróci zero ponieważ nie ma sposobu aby ciąg testowy mógłby być znaleziony w ciągu źródłowym w takiej sytuacji
2) Funkcja index działa jak następuje:
   i := 1
   while (i < (długość(źródło) – długość(test)) i
   test <> substr (źródło, i, długość (test) do
   i := i+1;

Kiedy ta pętla się koczy, jeśli (i < długość(źródło) –długość(test) wtedy zawiera indeks do źródła gdzie zaczyna się test. W przeciwnym razie test nie jest podciągiem źródła. Używając poprzedniego przykładu, pętla ta porównuje test ze źródłem w następujący sposób:

```
i = 1
test:       world                   Nie dopasowane
źródło:     Hello world

i = 2
test:        world                  Nie dopasowane
źródło:     Hello world

i = 3
test:         world                 Nie dopasowane
źródło:     Hello world

i = 4
test:          world                Nie dopasowane
źródło:     Hello world

i = 5
test:          world                Nie dopasowane
źródło:     Hello world

i = 6
test:          world                Nie dopasowane
źródło:     Hello world
```

i = 7
test: world Dopasowane
źródło: Hello world

Są (algorytmiczne) lepsze sposoby wykonania tego porównania, jednakże powyższy algorytm pozwala na stosowanie instrukcji ciągów 80x86 i jest bardzo łatwy do zrozumienia. Kod Index jest następujący:

```
;INDEX – oblicza offset jednego ciągu w innym
;
;Na wejściu:
;
;ES: DI                 Wskazują na ciąg testowy ,którego szuka INDEX w ciągu źródłowym
;
;DS:SI                  Wskazują na ciąg źródłowy który (przypuszczalnie) zawiera ciąg szukany
;                       przez INDEX
;
;Na wyjściu:
;
;AX                     Zawiera offset do ciągu źródłowego gdzie znaleziono ciąg testowy
;
INDEX       proc    near
            push    si
            push    di
            push    bx
            push    cx
            pushf                   ;zachowanie wartości flagi kierunku
            cld
            mov     al., es:[di]    ;pobranie długości ciągu  testowego
            cmp     al., [si]       ;zobacz czy jest dłuższy niż długość ciągu
            ja      NotThere        ;źródłowego

;Obliczenie indeksu ostatniego znaku potrzebnego do wyliczenia ciągu testowego w ciągu
;źródłowym
            mov     al., es:[di]    ;długość ciągu testowego
            mov     cl, al.         ;zachowanie na później
            mov     ch, 0
            sub     al., [si]       ;długość ciągu źródłowego
            mov     bl, al.         ;liczba powtórzeń pętli
            inc     di              ;przeskoczenie długości bajtu
            xor     ax, ax          ;inicjalizacja indeksu zerem
CmpLoop:    inc     ax              ;indeks na jeden
            inc     si              ;przeniesienie na olejny znak w źródle
            push    si              ;zachowani wskaźnika ciągu i długości ciągu testowego
            push    di
            push    cx
        rep cmpsb                   ;porównanie ciągów
            pop     cx              ;przywrócenie wskaźnika ciągu
            pop     di              ;i długości
            pop     si
            je      Foundindex      ;jeśli znaleziono podciąg
            dec     bl
            jnz     CmpLoop         ;spróbujmy następne wejście w ciągu źródłowym

;Jeśli  jesteśmy tutaj, ciąg testowy nie pojawił się wewnątrz ciągu źródłowego

NotThere:   xor     ax, ax          ;zwraca INDEX = 0

;Jeśli podciąg  został znaleziony w powyższej pętli usuwamy śmiecie pozostawione na stosie

Foundindex: popf
```

```
                pop     cx
                pop     bx
                pop     di
                pop     si
                ret
INDEX           endp
```

15.3.3 REPEAT

funkcja ciągów Repeat oczekuje trzech parametrów: adresu ciągu, długości i znaku. Konstruuje ona ciąg określonej długości zawierający „długość" kopiowanego , określonego znaku. Na przykład , Repeat (STR,5,'*') przechowa ciąg '*****' w zmiennej napisowej STR. Jest to bardzo łatwa do napisania, dzięki instrukcji stosb:

```
;REPEAT                 Konstruuje ciąg o długości CX, gdzie każdy element jest inicjalizowany
;                       znakiem przekazywanym w Al.
;
;Na wejściu:
;
;ES:DI                  Wskazują na ciąg do stworzenia
;CX                     Zawiera długość ciągu
;AL.                    Zawiera znak, którym będzie inicjalizowany każdy element ciągu
;

REPEAT          proc    near
                push    di
                push    ax
                push    cx
                pushf                   ;zachowanie wartości flagi kierunku
                cld
                mov     es:[di],cl      ;zachowanie długości ciągu
                mov     ch, 0
                inc     di              ;start ciągu od następnej lokacji
        rep     stosb
                popf
                pop     cx
                pop     ax
                pop     di
                ret
REPEAT          endp
```

15.3.4 INSERT

Funkcja ciągu Insert wprowadza jeden ciąg do innego. Oczekuje trzech parametrów, ciągu źródłowego, ciągu przeznaczenia i indeksu. Insert wprowadza ciąg źródłowy do ciągu przeznaczenia poczynając od offsetu określonego przez parametr indeksowy. HLL'e zazwyczaj wywołują procedurę Insert jak następuje:

```
Źródło := ' there';
Przezn := 'Hello world';
INSERT (źródło, przezn, 6);
```

Powyższe wywołanie Insert zmieni zawartość źródła, zawierając ciąg 'Hello there world'. Jest to zrobione przez wstawienie ciągu 'there' przed szóstym znakiem w 'Hello world'.

Procedura wstawiania używa następującego algorytmu::

Insert (Src, przezn, index) ;

1) Przenosi znaki z lokacji przezn + index bezpośrednio na koniec ciągu przeznaczenia length(Src) bajtów w pamięci
2) Kopiuje znaki z ciągu Src do lokacji przezn + index

3) Modyfikuje długość ciągu przeznaczenia więc jest to suma długości przeznaczenia i źródła. Poniższy kod implementuje ten algorytm:

```
;INSERT – Wprowadza jeden ciąg do innego
;
;Na wejściu:
;
;DS:SI  Wskazują ciąg źródłowy do wstawienia
;
;ES:DI  Wskazują ciąg przeznaczenia do którego będzie wstawiany ciąg źródłowy
;
;DX  zawiera offset w ciągu przeznaczenia gdzie ciąg źródłowy będzie wstawiony
;
;
;Wszystkie rejestry są  zachowane
;
;Warunek błędu
;
;Jeśli długość nowo stworzonego ciągu jest większa iż 255, operacja wstawiania nie będzie wykonana
;a flaga przeniesienia będzie ustawiona
;
;Jeśli indeks jest większy niż długość ciągu przeznaczenia, wtedy ciąg  źródłowy będzie dołączony do końca
;ciągu przeznaczenia

INSERT      proc    near
            push    si
            push    di
            push    dx
            push    cx
            push    bx
            push    ax
            clc                             ;zakładamy brak błędu
            pushf
            mov     dh, 0                   ;dla bezpieczeństwa

;Najpierw zobaczymy czy nowy ciąg nie będzie zbyt długi
            mov     ch, 0
            mov     ah, ch
            mov     bh, ch
            mov     al., es:[di]            ;AX = długość ciągu przeznaczenia
            mov     cl. [si]                ;CX = długość ciągu źródłowego
            mov     bl, al.                 ;BX = długość nowego ciągu
            add     bl, cl
            jc      TooLong                 ;przerwanie jeśli zbyt duży
            mov     es:[di], bl             ;uaktualnienie długości

;zobaczymy czy wartość index jest zbyt duża
            cmp     dl, al.
            jbe     IndexIsOK
            mov     dl, al.
IndexIsOK:

;Teraz zrobimy miejsce dla ciągu który będzie wstawiony
            push    si              ;zachowanie na później
            push    cx

            mov     si, di          ;SI wskazuje koniec bieżącego
            add     si, ax          ;ciągu przeznaczenia
            add     di, bx          ;DI wskazuje koniec nowego ciągu
            std
```

```
        rep     movsb                   ;otwarcie miejsca na nowy ciąg
;Teraz kopiujemy ciąg źródłowy do otwartej przestrzeni
                pop     cx
                pop     si
                add     si, cx          ;wskazanie końca ciągu źródłowego
        rep     movsb
                jmp     INSERTDone
TooLong::       popf
                stc
                pushf
INSERTDone:     popf
                pop     ax
                pop     bx
                pop     cx
                pop     dx
                pop     di
                pop     si
                ret
INSERT          endp
```

15.3.5 DELETE

Delete usuwa znaki z ciągu Oczekuje trzech parametrów – adres ciągu, indeksu do tego ciągu i liczby znaków do usunięcia z tego ciągu. HLL'e zazwyczaj wywołują Delete za pomocą :

Delete (Str , index, length)

Na przykład,

```
Str := 'Hello there world';
Delete(str, 7,6);
```

To wywołanie Delete pozostawi ciąg 'Hello world'. Algorytm dla operacji usuwania jest następujący:

1) Odejmujemy wartość parametru długości od długości ciągu przeznaczenia i uaktualniamy długość ciągu przeznaczenia tą nową wartością.
2) Kopiujemy kolejne znaki usuwanego ciągu nad krańcem usuwanego pod ciągu

Jest parę błędów, które mogą wystąpić kiedy używamy procedury usuwającej. Wartość indeksu może by zerem lub większa niż rozmiar określonego ciągu. W tym przypadku procedura Delete nie powinna robić niczego z tym ciągiem. Jeśli suma parametrów index i length jest większa niż długość ciągu, wtedy procedura Delete powinna usnąć wszystkie znaki do końca ciągu. Poniższy kod implementuje procedurę Delete

```
;DELETE – usuwa podciągi z ciągu
;
;Na wejściu:
;
;DS.:SI                 Wskazują ciąg źródłowy
;DX                     Indeks do ciągu zaczynającego podciąg do usunięcia
;
;CX                     Długość podciągu do usunięcia
;
;Warunki błędu
;
;Jeśli DX jest  większy niż długość ciągu, wtedy operacja jest przerywana
;
;Jeśli DX+CX są większe niż długość ciągu, DELETE usunie tylko te znaki z DX do końca  ciągu

DELETE          proc    near
                push    es
                push    si
                push    di
```

```
                    push    ax
                    push    cx
                    push    dx
                    pushf                       ;zachowanie flagi kierunku
                    mov     ax, ds.             ;ciągi przeznaczenia i źródłowy
                    mov     es, ax              ;są takie same
                    mov     ah, 0
                    mov     dh, ah              ;dla bezpieczeństwa
                    mov     ch, ah
;Zobaczymy czy istnieje warunek błędu
                    mov     al., [si]           ;pobranie długości ciągu
                    cmp     dl, al.             ;czy indeks jest zbyt długi
                    ja      TooBig
                    mov     al., dl             ;teraz zobaczmy czy INDEX + LENGTH
                    add     al., cl             ;są zbyt duże
                    jc      Truncate
                    cmp     al., [si]
                    jbe     LengthIsOK
;Jeśli podciąg jest zbyt duży obcinamy go do odpowiedniego

Truncate:           mov     cl, [si]            ,obliczamy maksymalną długość
                    sub     cl, dl
                    inc     cl
;Obliczamy długość nowego ciągu

LengthIsOK:         mov     al., [si]
                    sub     cl, cl
                    mov     [si], al.
;OK., teraz usuwamy określony podciąg
                    add     si, dx          ;obliczamy adres podciągu do usunięcia
                    mov     di, si          ;i adres pierwszego znaku po nim
                    add     di, cx
                    cld
                rep movsb                   ;usunięcie ciągu
TooBig:             popf
                    pop     dx
                    pop     cx
                    pop     ax
                    pop     di
                    pop     si
                    pop     es
                    ret
DELETE              endp
```

15.3.6 KONKATENCJA

Operacja konkatenacji bierze dwa ciągi i dołącza jeden na koniec drugiego. Na przykład, Concat(‘Hello ‘, ‘world’) stworzy łańcuch ‘Hello world’. Niektóre języki wysokiego poziomu traktują konkatenację jako wywołanie funkcji, inne jako wywołanie procedury. Ponieważ w asemblerze wszystko jest wywołaniem procedury, zaadoptujemy składnie proceduralną. Nasza procedura Concat przybierze postać jak następuje:

Concat(źróło1.źróło2, przeznaczenie)

Procedura ta skopiuje źródło1 do przeznaczenia, potem dołączy źródło2 na koniec przeznaczenia.

```
;Concat                 Kopiuje ciąg wskazywany przez SI do ciągu wskazywanego przez DI a
;                       potem łączy ciąg wskazywany przez BX do ciągu przeznaczenia
;
;Na wejściu
;
;DS:SI                      wskazuje na pierwszy ciąg źródłowy
;DS.:BX                     wskazuje na drugi ciąg źródłowy
```

```
;ES:DI                                  wskazuje na ciąg przeznaczenia
;
;Warunki błędu
;
;Suma długości dwóch ciągów jest większa niż 255. W takim razie, drugi ciąg będzie obcięty aby cały ciąg był
;mniejszy niż 256 znaków.

CONCAT          proc    near
                push    si
                push    di
                push    cx
                push    ax
                pushf
    ;Kopiujemy pierwszy ciąg do ciągu przeznaczenia:
                mov     al., [si]
                mov     cl, al.
                mov     ch, 0
                mov     ah, ch
                add     al., [bx]           ;oblicza sumę długości ciągów
                adc     ah, 0
                cmp     ax, 256
                jb      SetNewLength
                mov     ah,[si]             ,zachowanie oryginalnej długości ciągu
                mov     al., 255            ;ustalenie długości ciągu na 255
    SetNewLength:
                mov     es:[di], al.        ;zachowanie nowej długości ciągu
                inc     di                  ;przeskok długości bajtu
                inc     si
            rep movsb                       ;skopiowanie źróło1 do ciągu przeznaczenia
    ;Jeśli suma dwóch  ciągów jest zbyt duża, drugi ciąg musi być obcięty

                mov     cl, [bx]            ;pobrani długości drugiego ciągu
                cmp     ax, 256
                jb      LengthAreOK
                mov     cl, ah              ;obliczanie obciętej długości
                neg     cl                  ;CL := 256 – Length(Str1)

    LengthAreOK:
                lea     si, 1 bx]           ;wskazuje drugi ciąg i pominięcie
                                            ;długości ciągu
                cld
            rep movsb                       ;wykonanie konkatenacji

                popf
                pop     ax
                pop     cx
                pop     di
                pop     si
                ret
CONCAT          endp
```

15.4 FUNCKJE CIĄGÓW W BIBLIOTECE STNDARDOWEJ UCR

Standardowa Biblioteka UCR dostarcza bardzo bogatego zbioru funkcji ciągów jakie możemy użyć. Te podprogramy, przeważnie, trochę podobne do funkcji ciągów Standardowej Biblioteki C. Funkcje te wspierają ciągi zakończone zerem zamiast ciągów z przedrostkiem długości wspierane przez funkcje z poprzedniej sekcji.

Ponieważ jest wiele różnych podprogramów ciągów UCR StdLib, a kody źródłowe dla wszystkich tych podprogramów są public domain (i są zamieszczone na dołączonym do tego tekstu CD-ROM'ie), następne sekcje nie będą omawiały implementacji każdego z tych podprogramów. Zamiast tego poniższe sekcje skoncentrują się na tym jak użyć tych podprogramów bibliotecznych.

Biblioteka UCR często dostarcza kilu wariantów tego samego podprogramu. Ogólnie przyrostek „l", „m" lub „ml" pojawi się na końcu nazwy tych wariantów podprogramów. Przyrostek 'l' określa „stałą

literalną", Podprogram z przyrostkiem „l" (lub „ml") wymaga dwóch argumentów ciągów. Pierwszy jest wskazywany przez es:di a drugi występuje bezpośrednio po wywołaniu w strumieniu kodu.

Większość podprogramów ciągów StdLib dział n określonych ciągach (lub jednym z ciągów jeśli funkcja ma dwa argumenty). Przyrostek „m" (lub „ml") instruuje funkcję ciągu aby zaalokowała pamięć na stercie (używając malloc, stąd przyrostek „m") dla nowego ciągu i przechowała tam zmodyfikowany wynik zamiast zmieniać ciąg (i) źródłowy. Podprogramy te zawsze zwracają wskaźnik do nowo stworzonego ciągu w rejestrach es:di. W przypadku błędu alokacji pamięci (niewystarczająca pamięć) podprogramy te z przyrostkiem „m" lub „ml" zwracaj ustawioną flagę przeniesienia. Zwracają one wyzerowaną tą flagę jeśli operacja zakończyła się sukcesem.

15.4.1 STRBDEL, STRBDELM

Te dwa podprogramy usuwają czołowe spacje z ciągu StrBDel usuwa każdą spację czołową z ciągu wskazywanego przez es:di. W rzeczywistości modyfikuje ciąg źródłowy. StrBDelm robi kopię ciągu na stercie z usuniętymi spacjami czołowymi. Jeśli nie ma żadnych spacji czołowych, wtedy podprogram StrBDel zwróci oryginalny ciąg bez modyfikacji. Zauważmy, że te podprogramy wpływają tylko na spacje czołowe (te pojawiające się na początku ciągu). Nie usuwają spacji końcowych i spacji w środku ciągu. Do usuwania spacji końcowych zajrzyj po Strtrim. Przykład:

```
MyString        byte    „      Hello there, this is my string", 0
MyStrPtr        dword   MyString
                -
                -
                -
                les     di, MyStrPtr
                strbdelm                ;tworzy nowy ciąg bez czołowych spacji
                jc      error           ;wskaźnik do ciągu jest zwracany w ES:DI
                puts                    ;drukuje ciąg wskazywany przez ES:DI
                free                    ;dealokuje pamięć zaalokowaną przez  strbdelm

;Zauważmy, że „MyString" zawiera jeszcze czołowe spacje. Poniższe wywołanie printf wydrukuje ciąg wraz z
;tymi spacjami. Powyższe  „strbdelm" nie zmienia MyString
                printf
                byte    „MyString = '%s' \n", 0
                dword   MyString
                -
                -
                -
                les     di, MyStrPtr
                strbdel
;Teraz rzeczywiście usuwamy czołowe spacje z „MyString"
                printf
                byte    „MyString = '%s'\n", 0
                dword   MyString
                -
                -
                -
```

Dane wyjściowe tego fragmentu kodu:

```
Hello there, this is my string
MyString = '      Hello there, this is my string'
MyString = 'Hello there, this is my string'
```

15.4.2 STRCAT, STRCATL, STRCATM, STRCATML

Podprogramy strcat(xx) wykonują konkatenację ciągów. Na wejściu es:di wskazuje na pierwszy ciąg a dla strcat / strcatm dx:si wskazuje drugi ciąg. Dla strcatl i strcatlm drugi ciąg następuje po wywołaniu w strumieniu kodu. Podprogramy te tworzą nowy ciąg przez dołączenie drugiego ciągu na końcu pierwszego. W przypadku strcat i strcatl, drugi ciąg jest dołączany bezpośrednio do końca pierwszego ciągu (es:di) w pamięci. Musimy upewnić się ,że jest wystarczająca ilość pamięci przy końcu pierwszego ciągu, aby można było

dołączyć nowe znaki. Strcatm i strcatml tworzą nowy ciąg na stercie (używając malloc) przechowując tam wynik połączenia. Przykład:

```
String1         byte    „Hello „„ 0
                byte    16   dup (0)                ;miejsce dla konkatenacji
String2         byte    ‘World”, 0

;Poniższe makro ładuje ES:DI adresem określonym w argumencie

lesi            macro   operand
                mov     di , seg  operand
                mov     es, di
                mov     di, offset operand
                endm
;Poniższe makro ładuje DX:SI adresem  określonego operandu

ldxi            macro   operand
                mov     dx, seg operand
                mov     si, offset  operand
                endm
                -
                -
                -
                lesi    String1
                ldxi    String2
                strcatm                         ;Tworzy „Hello world”
                jc      error                   ;jeśli nie wystarczająca pamięć
                print
                byte    „strcatm: „„0
                puts                            ;Drukuje „Hello world”
                putcr
                free                            ;dealokacja pamięci ciągu
                -
                -
                -
                lesi    String1                 ;tworzenie ciągu
                strcatml                        ;’Hello world”
                jc      error                   ;jeśli niewystarczająca pamięć
                byte    „there”, 0
                print
                byte    „strcatml:”, 0
                puts                            ;drukuje „Hello world”
                putcr
                free
                -
                -
                -
                lesi    String1
                ldxi    String2
                strcat                          ;tworzenie „Hello world”
                printf
                byte    „strcat: %s\n”, 0
                -
                -
                -
;Notka: ponieważ strcat w rzeczywistości modyfikuje String1, poniższe wywołanie  strcatl dołączy
;”there” na koniec ciągu „Hello world”
                lesi    String1
                strcatl
                byte    „there”, 0
```

```
                printf
                byte    „strcatl: %s\n”, 0
                -
                -
                -
```

Powyższy kod stworzy następujące dane wyjściowe:

```
strcatm: Hello world
strcatml: Hello there
strcat: Hello world
strcatl: Hello world there
```

15.4.3 STRCHR

Strchr poszukuje pierwszego wystąpienia pojedynczego znaku wewnątrz ciągu .Jest w tym podobna trochę do instrukcji scasb. Jednak nie musimy określać wyraźnie długości kiesy używamy tej funkcji jak było to przy scasb.

Na wejściu es:di wskazują ciąg jaki chcemy przeszukać, al. Zawiera wartość szukana. Przy zwracaniu, flaga przeniesienia oznacza sukces (C = 1 znaczy, że znak nie jest obecny w ciągu C =0 znaczy, że znak jest w ciągu). Jeśli znak został znaleziony w ciągu, cx zawiera indeks do ciągu gdzie strchr ulokował znak. Zauważmy, że pierwszy znak ciągu jest pod indeksem zero. Więc strchr zwróci zero jeśli al pasuje do pierwszego znaku ciągu. Jeśli flaga przeniesienia jest ustawiona, wtedy wartość w cx nie ma znaczenia. Przykład:

```
;Zauważmy, że poniższy ciąg ma kropkę pod lokacją „HasPeroiod +24”

HasPeriod       byte    „This string has period.:, 0
                -
                -
                -
                lesi    HasPeriod           ;zobacz strcat dla definicji lesi
                mov     al., ‘.’            ;poszukiwanie kropki
                strchr
                jnc     GotPeriod
                print
                byte    „ Żadnej kropki w ciągu ”,cr,lf, 0
                jmp     Done
;Jeśli kropka jest znaleziona, wyprowadzamy offset do ciągu:

GotPeriod:      print
                byte    „Kropka znaleziona pod offsetem „, 0
                mov     ax, cx
                puti
                putcr
Done:
```

Ten kod tworzy dane wyjściowe;

```
Znaleziono kropkę pod offsetem 24
```

15.4.4 STRCMP, STRCMPL,STRICMP, STRICMPL

Podprogramy te porównują ciągi używając porządku leksykograficznego. Na wejściu do strcmp lub stricmp, es:di wskazuje pierwszy ciąg a dx:si wskazuje drugi ciąg. Strcmp porównuje pierwszy ciąg z drugim i zwraca wynik porównania we fladze rejestrów. Strcmpl działa w podobny sposób, z wyjątkiem tego, że drugi ciąg jest wywoływany w strumieniu kodu. Podprogramy stricmp i stricmpl różnią się od swoich odpowiedników w tym, że ignorują wielkość liter podczas porównania. Podczas gdy strcmp zwróciłby „nie równe” kiedy porównuje ”Strcmp” z „strcmp”, podprogramy stricmp (i stricmpl) zwrócą „równe’ ponieważ jedyną różnicą jest duża litera kontra mała litera. „i” w stricmp i stricmpl oznacza „ignoruj wielkość liter”. Przykład:

```
String1         byte    „Hello world’, 0
String2         byte    „hello world”, 0
String3         byte    „Hello there’, 0
                -
                -
                -
                lesi    String1
                ldxi    String2
                strcmp
                jae     IsGtrEql
                printf
                byte    „%s jest mniejszy niż %s\n’, 0
                dword   String1, String2
                jmp     Try1
IsGtrEql:       printf
                byte    „%s jest większy niż %s\n”, 0
                dword   String1, String2

Try1:           lesi    String2
                strcmpl
                byte    „hi world!”, 0
                jne     NotEql
                printf
                byte    „Hmm..., %s jest równe ‘hi world!’\n”, 0
                dword   String2
                jmp Try1

NotEql:         printf
                byte    „%s nie jest równe ‘hi world!’ \n”, 0
                dword   String2

Try1:           lesi    String1
                ldxi    String2
                stricmp
                jne     BadCmp
                printf
                byte    „Ignoruj wielkość liter. %s równe %s \n”, 0
                dword   String1, String2
                jmp     Tryil

BadCmp:         printf
                byte    „Wow, stricmp nie działa! %s <> %S\n”,0
                dword   String1, String2

Tryil:          lesi    String2
                stricmpl
                byte    „hELLO THERE”, 0
                jne     BadCmp2
                print
                byte    „Stricmpl zadziałał”, cr, lf, 0
                jmp     Done

BadCmp2:        print
                Byte    „Stricmp nie zadziałał”, cr, lf, 0
Done:
```

15.4.5 STRCPY,STRCPYL,STRDUP,STRDUPL

Podprogramy strcpy i strdup kopiują jeden ciąg do innego. Nie ma podprogramów strcpym lub strcpyml . Strdup i strdupl odpowiadają tym operacjom. Standardowa Biblioteka UCR używa nazw strdup i strdupl zamiast strcpym i strcpyml, więc używa tych samych nazw jak standardowa biblioteka C.

Strcpy kopiuje ciąg wskazywany przez es:di do komórki pamięci zaczynającej się pod adresem dx:si. Nie ma żadnej kontroli błędów; musimy zapewnić, że jest odpowiednia ilość wolnej przestrzeni pod lokacją dx:si przed wywołaniem strcpy. Strcpy zwraca pod es:di wskazujący ciąg przeznaczenia (to znaczy oryginalną wartość dx:si). Strcpyl działa w podobny sposób, z wyjątkiem ciągu źródłowego następującego po wywołaniu.

Strdup duplikuje ciąg , który wskazuje es:di i zwraca wskaźnik do nowego ciągu na stercie. Strdupl działa w podobny sposób, z wyjątkiem tego, że ciąg występuje po wywołaniu. Jak zwykle, flaga przeniesienia jest ustawiona jeśli wystąpi błąd alokacji pamięci kiedy używamy strdup lub strdupl. Przykład:

```
String1         byte    „Copy this string”, 0
String2         byte    32   dup (0)
String3         byte    32   dup (0)
StrVar1         dword   0
StrVar2         dword   0
                -
                -
                -
                lesi    String1
                ldxi    String2
                strcpy

                ldxi    String3
                strcpyl
                byte    „This  string, too!”, 0

                lesi    String1
                strdup
                jc      error                   ;jeśli niewystarczająca pamięć
                mov     word  ptr  StrVar1, di  ;zachowanie wskaźnika do ciągu
                mov     word  ptr  StrVar1+2, es

                strdupl
                jc      error
                byte    „Also this string”, 0
                mov     word   ptr StrVar2, di
                mov     word   ptr StrVar2+2, es

                printf
                byte    „strcpy: %s\n”
                byte    „strcpyl: %s\n”
                byte    „strdup: %^s\n”
                byte    „strdupl: %^s\n”, 0
                dword   String2, String3, StrVar1, StrVar2
```

15.4.6 STRDEL, STRDELM

Strdel i strdelm usuwają znaki z ciągu. Strdel usuwa określone znaki z ciągu , strdelm tworzy nową kopię ciągu źródłowego bez określonych znaków. Na wejściu, es:di wskazują ciąg do obróbki, cx zawiera indeks do ciągu gdzie zaczyna się usuwanie a ax zawiera liczbę znaków do usunięcia z ciągu. Przy zwrocie, es:di wskazują nowy ciąg (który jest na stercie jeśli wywołaliśmy strdelm). Tylko dla strdelm ,jeśli flaga przeniesienia jest ustawiona przy zwrocie, będzie błąd alokacji pamięci. Tak jak dla wszystkich podprogramów ciągów StdLib UCR, wartości indeksu dla ciągów są oparte na zerze. To znaczy zero jest indeksem pierwszego znaku w ciągu źródłowym. Przykład:

```
String1         byte    „Hello there, how are you?”, 0
                -
                -
                -
```

```
                lesi    String1
                mov     cx, 5                   ;start od pozycji pięć ( „ there”)
                mov     ax, 6                   ;usunięcie szóstego znaku
                strdelm                         ;stworzenie nowego ciągu
                jc      error                   ;jeśli niewystarczająca pamięć
                print
                byte    „New string:”, 0
                puts
                putcr

                lesi    String1
                mov     ax, 11
                mov     cx, 13
                strdel
                printf
                byte    „Zmodyfikowany ciąg; %s\n”, 0
                dword   String1
```

Kod ten daje nam co następuje:

```
New string: Hello, how are you?
Modified string: Hello there
```

15.4.7 STRINS, STRINSL, STRINSM, STRINSML

Funkcje strins(xx) wprowadzają jeden ciąg do innego. Dla wszystkich czterech podprogramów es:di wskazuje ciąg źródłowy do którego chcemy wprowadzić inny ciąg .Cx zawiera punkt wprowadzenia (0...długość ciągu źródłowego) Dla strins i strinsm, dx:si wskazuje ciąg jaki życzymy sobie wprowadzić. Dla strinsl i strinslm ciąg do wprowadzenia pojawia się jako stałą literalna w strumieniu kodu. Strins i strinsl wprowadzają drugi ciąg bezpośrednio do ciągu wskazywanego przez es:di. Strinsm i strinsml robią kopię ciągu źródłowego i wprowadzają drugi ciąg do tej kopii. Zwracają one wskaźnik do nowego ciągu w es:di. Jeśli wystąpi błąd alokacji pamięci wtedy strinsm / strinsml ustawiają flagę przeniesienia przy powrocie. Dla strins i strinsl, pierwszy ciąg musi mieć odpowiednio zaalokowaną pamięć dla przechowania nowego ciągu. Przykład:

```
InsertInMe      byte    „Insert >< Here”, 0
                byte    16   dup (0)
InsertStr       byte    „insert this’, 0
StrPtr1         dword   0
StrPtr2         dword   0
                -
                -
                -
                lesi    InsertInMe
                ldxi    InsertStr
                mov     cx, 8                   ;wstaw przed „<”
                strinsm
                mov     word  ptr  StrPtr1, di
                mov     word   ptr  StrPtr1+2, es

                lesi    InsertInMe
                mov     cx, 8
                strinsml
                byte    „insert that’, 0
                mov     word ptr StrPtr2, di
                mov     word  ptr StrPtr2+2, es

                lesi    InsertInMe
                mov     cx, 8
                strinsl
                byte    „  „„ 0                         ;dwie spacje
```

```
                lesi    InsertInMe
                ldxi    InsertStr
                mov     cx, 9                           ;przed pierwszą ze spacji
                strins

                printf
                byte    „Pierwszy ciąg:  %^s\n”
                byte    „Drugi ciąg: %^s\n”
                byte    „Trzeci ciąg:  %s\n”, 0
                dword   StrPtr1, StrPtr2, InsertInMe
```

Zauważmy, że powyższe operacje strins i strinsl, obie wprowadzają ciągi do tego samego ciągu przeznaczenia. Dane wyjściowe powyższego kodu:

Pierwszy ciąg: Insert >insert this< here
Drugi ciąg: Insert> insert that < here
Trzeci ciąg: Insert > insert this < here

15.4.8 STRLEN

Strlen oblicza długość ciągu wskazywanego przez es:di. Zwraca liczbę znaków aż do, ale nie wliczając, bajtu zakończonego zerem. Zwraca tą długość w rejestrze cx. Przykład:

```
GetLen          byte            „this string is 33 characters long”, 0
                -
                -
                -
                lesi    GetLen
                strlen
                print
                byte    „this string is „, 0
                mov     ax, cx                  ;puti potrzebuje długości w AX!
                puti
                print
                byte    „characters long”, cr, lf, 0
```

15.4.9 STRLWR, STRLWRM,STRUPR,STRUPRM

Strlwr i Strlwrm konwertują dużej litery w ciągu na litery małe. Strupr i Struprm konwertują małe litery w ciągu na litery duże. Podprogramy te nie wpływają na żadne inne znaki obecne w ciągu. Dla wszystkich czterech podprogramów, es:di wskazują ciąg źródłowy do konwersji. Strlwr i strupr modyfikują znaki bezpośrednio w ciągu. Strlwrm i struprm robią kopię ciągu na stercie a potem konwertują znaki w nowym ciągu. Zwracają one również wskaźnik do tego nowego ciągu w es:di. Jak zwykle przy podprogramach StdLib UCR, strlwrm i struprm zwracają ustawioną flagę przeniesienia jeśli wystąpi błąd alokacji pamięci. Przykład:

```
String1         byte    „This string has lower case.” , 0
String2         byte    „THIS STRING has Upper Case.” , 0
StrPtr1         dword   0
StrPtr2         dword   0
                -
                -
                -
                lesi    String1
                struprm                         ;konwersja małych liter na duże
                jc      error
                mov     word  ptr  StrPtr1, di
                mov     word  ptr StrPtr1+2, es
```

```
                lesi    String2
                strlwrm                         konwersja dużych liter na małe litery
                jc      error
                mov     word   ptr StrPtr2, di
                mov     word  ptr StrPtr2+2, es

                lesi    String1
                strlwr                          ;konwersja na małe litery, na miejscu

                lesi    String2
                strupr                          ;konwersja na duże litery, w miejscu

                printf
                byte    „struprm: %^s\n”
                byte    „strlwrm: %^s\n”
                byte    „strlwr: %s\n”
                byte    „strupr: %s\n”, 0
                dword   StrPtr1, StrPtr2, String1, String2
```

Powyższy fragment drukuje co następuje:

```
struprm: THIS STRING HAS LOWER CASE
strlwrm: this string has upper case
strlwr: this string has lower case
strupr: THIS STRING HAS UPPER CASE
```

15.4.10 STRREV, STRREVM

Te dwa podprogramy odwracają znaki w ciągu. Na przykład jeśli przekażemy do strrev ciąg „ABCDEF” skonwertuje go do ciągu „FEDCBA” .Jak możemy oczekiwać, podprogram strrev odwraca ciąg którego adres przekazujemy w es:di; strrevm najpierw robi kopię ciągu na stercie i odwraca znaki pozostawiając niezmieniony ciąg oryginalny. Oczywiście strrevm zwróci ustawioną flagę przeniesienia jeśli wystąpi błąd alokacji pamięci. Przykład:

```
Palindrome      byte    „radar”, 0
NotPaldrm       byte    „x+y – z”, 0
StrPtr1         dword   0
                -
                -
                -
                lesi    Palindrome
                strrevm
                jc      error
                mov     word   ptr StrPtr1, di
                mov     word   ptr StrPtr1+2, es

                lesi    NotPaldrm
                strrev

                printf
                byte    „First string: %^s\n”
                byte    „Second string: %s\n”, 0
                dword   StrPtr1, NotPaldrm
```

Powyższy kod da nam takie dane wyjściowe:

```
First string: radar
Second string: z – y+x
```

15.4.11 STRSET, STRSETM

Strset i strsetm replikują pojedynczy znak w całym ciągu . ich zachowanie nie jest jednak całkiem podobne. W szczególności podczas gdy strsetm jest trochę podobna do funkcji repeat, strset nie. Oba podprogramy oczekują wartości pojedynczego znaku w rejestrze al. Replikują ten znak w całym ciągu .Strsetm wymaga również licznika w rejestrze cx. Tworzy na stercie ciąg składający się z cx znaków i zwraca wskaźnik do tego ciągu w es:di (zakładając, ze nie wystąpi żaden błąd). Strset, z drugiej strony, oczekuje przekazania mu adresu istniejącego ciągu w es:di. Zamienia on każdy znak w tym ciągu ze znakiem w al. Zauważmy, że nie określamy długości kiedy używamy funkcji strset, strset używa długości istniejącego ciągu. Przykład:

```
String1         byte    „Hello there”, 0
                -
                -
                -
                lesi    String1
                mov     al., ‘*’
                strset

                mov     cx, 8
                mov     al., ‘#’
                strsetm

                print
                byte    „String2: „„0
                puts
                printf
                byte    „\nString1: %s\n”, 0
                dword   String1
```

Powyższy kod da nam dane wyjściowe takie:

```
String2: ########
String1: **********
```

15.4.12 STRSPAN. STRSPANL, STRCSPAN, STRCSPANL

Te cztery podprogramy szukają w całym ciągu znaku który jest albo w jakimś określonym zbiorze znaków (strspan, strspanl) lub nie jest członkiem jakiegoś zbioru znaków (strcspan, strcspanl) . Te podprogramy pojawiły się w Bibliotece Standardowej UCR tylko dlatego, że pojawiają się w bibliotece standardowej C. Rzadko powinniśmy używać tych podprogramów. Biblioteka Standardowa UCR zawiera inne podprogramy do manipulowania zbiorami znaków i wykonywania operacji dopasowywania znaków. Pomimo to te podprogramy są czasami użyteczne i warte zajęcia się nimi tutaj.

Te podprogramy oczekują przekazania im adresów dwóch ciągów: ciągu źródłowego i ciągu zbioru znaków. Oczekują adresu ciągu źródłowego w es:di. Strspan i strcspan chcą adresu ciągu zbioru znaków w dx:si; ciąg zbioru znaków następuje po wywołaniu strspanl i strcspanl. Przy zwracaniu , cx zawiera indeks do ciągu, zdefiniowanego jak następuje:

strspan, strspanl:	indeks pierwszego znaku w źródle znalezionego z zbiorze znaków
strcspan, strcspanl:	indeks pierwszego znaku w źródle nie znalezionego w zbiorze znaków

Jeśli wszystkie znaki są w zbiorze (lub nie są w zbiorze) wtedy cx zawiera indeks do ciągu zakończonego zerem. Przykład:

```
Source          byte    „ABCDEFG  0123456”, 0
Set1            byte    „ABCDEFGHIJKLMNOPQRSTUVWXYZ”, 0
Set2            byte    „0123456789”, 0
Index1          word    ?
Index2          word    ?
Index3          word    ?
Index4          word    ?
                -
                -
```

```
                -
                lesi    Source
                ldxi    Set1
                strspan                         ;szukanie pierwszego znaku alfabetu
                mov     Index1, cx              ;indeks pierwszego znaku alfabetu

                lesi    Source
                lesi    Set2
                strspan                         ;szukanie pierwszego znaku liczbowego
                mov     Index2, cx

                lesi    Source
                strcspanl
                byte    „ABCDEFGHIJKLMNOPQRSTUVWXYZ”, 0
                mov     Index3, cx

                lesi    Set2
                strcspanl
                byte    „0123456789”, 0
                mov     Index4, cx

                printf
                byte    „First alpha char in Source is at offset %d\n”
                byte    „First numeric char is at offset  %d\n”
                byte    „First non-alpha in Source is at offset  %d\n”
                byte    „First non-numeric in Set2 is at offset  %d\n”
                dword   Index1, Index2, Index3, Index4
```

Kod ten da dane wyjściowe takie:

```
First alpha char in Source is at offset  0”
First numeric char is at offset  8
First non-alpha in Source is at offset  7
First non-numeric in Set2 is at offset 10
```

15.4.13 STRSTR, STRSTRL

Strstr poszukuje pierwszego wystąpienia jednego ciągu wewnątrz innego. Es:di zawiera adres ciągu w którym chcemy poszukać drugiego ciągu. Dx:si zawiera adres drugiego ciągu dla podprogramu strstr, strstrl przeszukuje drugi ciąg bezpośrednio występujący po wywołaniu w strumieniu kodu.

Przy powrocie z strstr lub strstrl, flaga przeniesienia będzie ustawiona jeśli drugi ciąg nie jest obecny w ciągu źródłowym. Jeśli flaga ta jest wyzerowana, wtedy drugi ciąg jest obecny w ciągu źródłowym a cx bezie zawierał (oparty o zero) indeks gdzie drugi ciąg został znaleziony. Przykład:

```
SourceStr       byte    „Search for ‘this’ in this string”, 0
SearchStr       byte    „this”, 0
                -
                -
                -
                lesi    SourceStr
                ldxi    SearchStr
                strstr
                jc      NotPresent
                print
                byte    „Found  string at offset  „, 0
                mov     ax, cx                  ;potrzebny offset in AX dla puti
                puti
                putcr

                lesi    SourceStr
                strstrl
```

```
                        byte    „for”, 0
                        jc      NotPresent
                        print
                        byte    „Found  ‘for’ at offset „, 0
                        mov     ax, cx
                        puti
                        putcr
NotPresent:
```

Powyższy kod wydrukuje co następuje:

```
Found string at offset  12
Found ‘for’ at offset  7
```

15.4.14 STRTRIM, STRTRIMM

Te dwa podprogramy są trochę podobne do strbdel i strbdelm. Zamiast usuwania czołowych spacji, obcinają końcowe spacje z ciągu. Strtrim obcina każdą końcową spację bezpośrednio w określonym ciągu w pamięci. Strtimm najpierw kopiuje ciąg źródłowy a potem obcina spacje w pamięci. Oba podprogramy oczekują przekazania adresu ciągu źródłowego w es:di. Strtrimm zwraca wskaźnik do nowego ciągu (jeśli może go zaalokować) w es:di. Zwraca również ustawione przeniesienie lub wyzerowanie do oznaczenia błąd/ brak błędu. Przykład:

```
String1         byte    „Space at the end       „ , 0
String2         byte    „   Space on both sides    „ , 0
StrPtr1         dword   0
StrPtr2         dword   0
                -
                -
                -
;TrimSpcs obcina spacje z obu końców ciągu . Zauważmy ,że jest to trochę bardziej wydajne wykonanie
;najpierw strbdel a potem strtrim. Ten podprogram tworzy nowy ciąg na stercie i zwraca wskaźnik  do tego
;ciągu w ES:DI

TrimSpcs        proc
                strbdelm
                jc      BadAlloc                ;zwracany jeśli błąd
                strtrim
                clc
BadAlloc:       ret
TrimSpcs        endp
                -
                -
                -
                lesi    String1
                strtrimm
                jc      error
                mov     word  ptr  StrPtr1, di
                mov     word  ptr  StrPtr1+2, es

                lesi    String2
                call    TrimSpcs
                jc      error
                mov     word  ptr  StrPtr2, di
                mov     word  ptr  StrPtr2+2, es

                printf
                byte    „First  string: ‘%s’\n”
                byte    „Second  string: ‘%s’\n”, 0
```

```
                dword   StrPtr1, StrPtr2
```

Kod daje wynik następujący:
First string: ‘Spaces at the end”
Second string: „Spaces on both sides”

15.4.15 INNE PODPROGRAMY CIĄGÓW W BIBLIOTECE STANDARDOWEJ UCR

Oprócz podprogramów strxxx wypisanych w tej sekcji, jest wiele dodatkowych podprogramów ciągów dostępnych w Bibliotece Standardowej UCR. Podprogramy do konwersji typów numerycznych (całkowite, hex, rzeczywiste) do ciągów lub vice versa, podprogramy dopasowania do wzorca i zbiór znaków i wiele innych konwersji. Podprogramy opisane w tym rozdziale są to te, których definicje pojawiają się w pliku nagłówkowym „strings.a” . Po więcej szczegółów o innych podprogramach ciągów zajrzyj do odnośnych części Biblioteki Standardowej UCR.

15.5 PODPROGRAMY ZBIORU ZNAKÓW W BIBLIOTECE STANDARDOWEJ UCR

Standardowa Biblioteka UCR dostarcza szerokiej kolekcji podprogramów zbioru znaków. Podprogramy te pozwalają nam stworzyć zbiór, wyzerować zbiór (ustawić je na pusty zbiór), dodawać i usuwać jedną lub więcej pozycji , przetestować członków zbioru , skopiować zbiór, obliczać sumę, iloczyn i różnice i wyciąganie pozycji ze zbioru, Chociaż przeznaczone go manipulowania zbiorami znaków, możemy użyć podprogramów zbioru znaków StdLib do manipulowania każdym zbiorem z 256 lub mniejszą ilością pozycji.

Pierwszą rzeczą do odnotowania o zbiorach StdLib jest ich format pamięciowy. 256 bitowa tablica zazwyczaj używa 32 kolejnych bajtów. Z powodu wydajności, zbiory formatów Standardowej biblioteki upakowywują osiem oddzielnych zbiorów do 272 bajtów (256 bajtów dla ośmiu zbiorów plus 16 bajtów górnych). Dla deklaracji zmiennych zbioru w segmencie danych powinniśmy użyć makra set. Makro to przybiera postać:

```
                set     SetName1, SetName2...., SetName8
```

Setname1...SetName8 przedstawia nazwy do ośmiu zmiennych zbioru. Możemy mieć mniej osiem nazw w polu
.
operandu, ale robiąc tak zmarnujemy kilka bitów w ustawieniu tablicy

Podprogram CreateSets dostarcza innego mechanizmu dla tworzenia zmiennych zbioru. W odróżnieniu od makra set, którego używaliśmy do tworzenia zmiennej zbioru w segmencie danych, podprogram CreateSets alokuje pamięć dla ośmiu dynamicznych zbiorów w czasie wykonania. Zwraca wskaźnik do pierwszej zmiennej zbioru w es:di. Pozostałe siedem zbiorów następuje w lokacjach es:di+1, es:di+2....., es:di+7. Typowy program, który alokuje zmienne zbioru dynamicznie może mieć taki kod:

```
Set0            dword   ?
Set1            dword   ?
Set2            dword   ?
Set3            dword   ?
Set4            dword   ?
Set5            dword   ?
Set6            dword   ?
Set7            dword   ?
                -
                -
                -
                CreateSets
                mov     word  ptr  Set0+2, es
                mov     word  ptr  Set1+2, es
                mov     word  ptr  Set2+2, es
                mov     word  ptr  Set3+2, es
                mov     word  ptr  Set4+2, es
                mov     word  ptr  Set5+2, es
                mov     word  ptr  Set6+2, es
                mov     word  ptr  Set7+2, es

                mov     word  ptr  Set0, di
                inc     di
```

```
        mov     word  ptr  Set1,di
        inc     di
        mov     word  ptr  Set2, di
        inc     di
        mov     word  ptr  Set3, di
        inc     di
        mov     word  ptr  Set4, di
        inc     di
        mov     word  ptr  Set5, di
        inc     di
        mov     word  ptr  Set6, di
        inc     di
        mov     word  ptr  Set7, di
        inc     di
```

Ten fragment kodu tworzy osiem różnych zbiorów na stercie, wszystkie puste, i przechowuje wskaźnik do nich w stosownej zmiennej wskaźnikowej.

Plik SHELL.ASM dostarcza skomentowanej lini kodu w segmencie danych, która zawiera plik STDSETS.A. Ten plik dostarcza definicji dla ośmiu powszechnie stosowanych zbirów znaków. Są to alpha (duże i małe litery alfabetu), lower (małe litery alfabetu), upper (duże litery alfabetu), digits ((„0”...”9”), xdigits („0”...”9”, „A”...”F”, i „a”...”f”), alphanum (duże i małe litery plus cyfry), whitespace (spacja, tabulator, powrót karetki, przesuniecie o jedną linię) i delimiters (białe znaki plus przecinki, średniki, mniejsze niż ,większy niż i pasek poziomy). Jeśli chcielibyśmy użyć tych standardowych zbiorów znaków w naszym programie, musimy usunąć średniki z początku instrukcji include w pliku SHELL.ASM.

Standardowa Biblioteka UCR dostarcza 16 podprogramów zbioru znaków: CreateSets, EmptySet, RangeSet, AddStr, AddStrl , RmvStr, RmvStrl, AddChar, RmvChar,Member,CopySet, SetUnion, SetIntersect, SetDifference, NextItem i RmvItem. Wszystkie te podprogramy z wyjątkiem CreateSets wymagają wskaźnika do zmiennej zbioru znaków w rejestrach es:di. Określone podprogramy mogą wymagać również innych parametrów.

Podprogram EmptySet zeruje wszystkie bity w zbiorze tworząc zbiór pusty. Podprogram ten wymaga adresu zmiennej zbioru w es:di. Poniższy przykład zeruje zbiór wskazywany przez Set1:

```
        les     si, Set1
        EmptySet
```

RangeSet łączy ,w pewnym zakresie, wartości w zmiennej zbioru wskazywanej przez es:di. Rejestr al. zawiera dolną granicę zakresu pozycji, ah zawiera górną granicę. Zauważmy, że al. musi być mniejszy niż lub równy ah. Poniższy przykład konstruuje zbiór wszystkich znaków sterujących (kody ASCII od jeden do 31, znak null [kod ASCII zero] nie jest ujęty w tym zbiorze):

```
        les     di, CtrlCharSet             ;wskaźnik do ctrl char set
        mov     al., 1
        mov     ah, 31
        RangeSet
```

AddStr i AddStrl dodają wszystkie znaki w ciągu zakończonym zerem zbiór znaków. Dla AddStr para rejestrów dx:si wskazuje ciąg zakończony zerem. Dla AddStrl ciąg zakończony występuje po wywołaniu AddStrl w strumieniu kodu. Podprogramy te łączą każdy znak określonego ciągu w zbiór. Poniższy przykład dodaje cyfry i znaki specjalne w zbiór FPDigits:

```
Digits          byte    „0123456789”, 0
                set     FPDigitsSet
FPDigits        dword   FPDigitsSet
                -
                -
                -
                ldxi    Digits              ;ładuje DX:SI adresem Digits
                les     di, FPDigits
                AddStr
                -
                -
                -
```

```
les     di, FPDigits
AddStrl
byte    „Ee.+-„, 0
```

RmvStr i RmvStrl usuwają znaki ze zbioru. Znaki dostarczamy w ciągu zakończonym zerem. Dla RmvStr, dx:si wskazuje ciąg znaków do usunięcia z ciągu. Dla RmvStrl, ciąg zakończony znakiem występuje po wywołaniu. Poniższy przykład używa RmvStrl usuwa specjalne symbole z FPDigits:

```
les     di, FPDigits
RmvStrl
byte    „Ee.+-„, 0
```

Podprogramy AddChar i RmvChar pozwalają nam dodawać lub usuwać pojedyncze znaki. Jak zwykle, es:di wskazuje zbiór; rejestr al zawiera znak jaki życzymy sobie dodać lub usunąć ze zbioru. Poniższy przykład dodaje spację do zbioru FPDigits i usuwa znak „„” (jeśli jest):

```
les     di, FPDigits
mov     al., ‘ ‘
AddChar
-
-
-
les     di, FPDigits
mov     al., ‘,’
RmvChar
```

Funkcja Member sprawdza czy znak jest obecny w zbiorze. Na wejściu es:di musi wskazywać zbiór a al musi zawierać znak do sprawdzenia. Na wyjściu, flaga zera jest ustawiana jeśli znak jest zawarty w zbiorze, flaga ta będzie wyzerowana jeśli znak nie należy do tego zbioru. Poniższy przykład odczytuje znaki z klawiatury dopóki użytkownik nie naciśnie klawisza, który jest białym znakiem

```
SkipWS:     get                 ;odczyt znaku od użytkownika do AL.
            lesi    WhiteSpace  ;adres zbioru WS w es:di
            member
            je      SkipWS
```

Podprogramy CopySet, SetUnion, SetIntersect i SetDifference działają na dwóch zbiorach znaków. Rejestry es:di wskazują zbiór znaków przeznaczenia, rejestry dx:si wskazują źródłowy zbiór znaków. CopySet kopiuje bity ze zbioru źródłowego do zbioru przeznaczenia, zamieniając oryginalne bity w zbiorze przeznaczenia. SetUnion oblicza sumę dwóch zbiorów i przechowuje wynik w zbiorze przeznaczania. SetIntersect oblicza iloczyn logiczny zbiorów i przechowuje wynik w zbiorze przeznaczenia. W końcu podprogram SetDifference oblicza DestSet : = DestSet – SrcSet.

Podprogramy NextItem i RmvItem pozwalają nam wyodrębnić elementy ze zbioru. NextItem zwraca w al. kod ASCII pierwszego znaku znalezionego w zbiorze. RmvItem robi to samo z tym, że usuwa ten znak ze zbioru. Podprogramy te zwracają zero w al. jeśli zbiór jest pusty (zbiory StdLib nie mogą zawierać znaku NULL) Możemy użyć podprogramu RmvItem do zbudowania podstawowego iteratora dla zbioru znaków.

Podprogramy zbiorów znaków Standardowej Biblioteki UCR są bardzo mocne. Z nimi możemy łatwo manipulować danymi ciągów znaków, zwłaszcza kiedy poszukujemy różnych wzorców wewnątrz ciągów. Będziemy rozpatrywać te podprogramy później kiedy będziemy się zajmować później w tym tekście dopasowywaniem do wzorca.

15.6 UZYWANIE INSTRUKCJI CIAGÓW Z INNYMI TYPAMI DANYCH

Instrukcje ciągów działają z innymi typami danych niż tylko ciągi znaków. Możemy użyć instrukcji ciągów do kopiowania całych tablic z jednej zmiennej do innej, inicjalizowania dużych struktur danych pojedynczą wartością lub do porównywania całych struktur danych dal równości lub nierówności. Jeśli kiedyś będziemy działać na strukturach danych zawierających kilka bajtów ,możemy zastosować instrukcje ciągów.

15.6.1 CIAGI CAŁKOWITE O WIELOKTOTNEJ PRECYZJI

Instrukcja cmps jest użyteczna przy porównaniu (bardzo) dużych wartości całkowitych. W odróżnieniu od ciągów znaków nie możemy porównywać liczb całkowitych cmps od najmniej znaczącego bajtu do bajtu

najbardziej znaczącego. Zamiast tego musimy porównywać je od bardziej znaczącego bajtu w dół do bajtu najmniej znaczącego. Poniższy kod porównuje dwie 12 bajtowe wartości całkowite:

```
                lea     di, integer1+10
                lea     si, integer2+10
                mov     cx, 6
                std
        repe    cmpsw
```

Po wykonaniu instrukcji cmpsw, flagi będą zawierały wynik porównania.

Możemy łatwo przypisać jeden długi ciąg całkowity do innego przy zastosowaniu instrukcji movs. Nic skomplikowanego, po prostu ładujemy rejestry si, di i cx i mamy to. Możemy wykonać inne operacje, wliczając w to operacje arytmetyczne i logiczne używając metod poszerzonej precyzji opisanej w rozdziale o operacjach arytmetycznych.

15.6.2 DZIAŁANIE Z CAŁYMI TABLICAMI I REKORDAMI

Jedynymi operacjami które stosujemy, generalnie , do wszystkich struktur tablicowych i rekordowych są przydzielanie i porównywanie (tylko dla równość / nierówność). Możemy zastosować instrukcji movs i cmps dla tych działań.

Działania takie jak dodawanie skalarne, transpozycje itp. mogą być łatwo zsyntetyzowane przy zastosowaniu instrukcji lods i stos. Poniższy kod pokazuje jak łatwo można dodać wartość 20 do każdego elementu tablicy całkowitej A:

```
                lea     si, A
                mov     di, si
                mov     cx, SizeOfA
                cld
AddLoop:        lodsw
                add     ax, 20
                stosw
                loop    AddLoop
```

Możemy zaimplementować inne operacje w podobny sposób

15.10 PODSUMOWANIE

80x86 dostarcza potężnego zbioru instrukcji ciągów. Jednak instrukcje te są bardzo prymitywne, użyteczne głównie do manipulowania blokami bajtów. Nie odpowiadają one instrukcjom ciągów jakie można znaleźć w językach wysoko poziomowych, Możemy jednak użyć instrukcji ciągów 80x86 do zsyntetyzowania tych funkcji normalnie powiązanych z HLL'ami. Rozdział ten wyjaśnia jak zbudować wiele z bardzo popularnych funkcji ciągów. Oczywiście głupotą jest stale odkrywanie koła, więc rozdział ten również opisuje wiele funkcji ciągów dostępnych w Bibliotece Standardowej UCR.

Instrukcje ciągów 80x86 dostarczają podstaw dla wielu operacji na ciągach pojawiających się w tym rozdziale. Dlatego też rozdział ten zaczyna się od przeglądu i szczegółowego omówienia instrukcji ciągów 80x86: przedrostków powtórzenia i flagi kierunku. Rozdział ten omawia działanie każdej instrukcji ciągu i opisuje jak możemy jej użyć do wykonania odnośnych zadań. Aby zobaczyć jak działają instrukcje ciągów zobacz:

- „Instrukcje ciągów 80x86"
- „Jak działają instrukcje ciągów"
- „Przedrostki REP/REPE/REPZ i REPNZ/REPNE"
- „Flaga kierunku"
- „Instrukcja MOVS'
- „Instrukcja CMPS
- „Instrukcja SCAS"
- „Instrukcja STOS"
- Instrukcja LODS"
- „Budowanie złożonych funkcji ciągów z LODS i STOS"
- „Przedrostki i instrukcje ciągów"

Chociaż Intel nazwał je „instrukcjami ciągów” w rzeczywistości nie działają one na abstrakcyjnych typach danych, jak zwykle myślimy o ciągach znaków. Instrukcje ciągów po prostu manipulują tablicami bajtów, słów lub podwójnych słów. Niestety nie ma pojedynczej definicji ciągu znaków co, bez wątpienia, jest powodem, ze nie ma specjalnych instrukcji w zbiorze instrukcji 80x86. Dwa z najbardziej popularnych typów ciągów znaków to ciągi z przedrostkiem długości i ciągi zakończone zerem, których używają, odpowiednio, Pascal i C

- „Ciągi znaków”
- „Typy ciągów”

Ponieważ zdecydowaliśmy się określić typ danych dla naszych ciągów znaków, następnym krokiem jest implementacja różnych funkcji dla przetwarzania tych ciągów. Rozdział ten dostarcza przykładów kilu różnych funkcji ciągów stworzonych specjalnie dla ciągów z przedrostkiem długości. Nauczmy się o tych funkcjach i zobaczmy kod ,który je implementuje przeglądając następujące sekcje:

- „Przypisywanie ciągów”
- „Porównywanie ciągów”
- „Funkcje ciągów znaków”
- „Substr”
- „Index”
- „Repeat”
- „Insert”
- „Delete”
- „Konkatenacja”

Biblioteka Standardowa UCR dostarcza bardzo bogatego zbioru funkcji ciągów specjalnie stworzonych dla ciągów zakończonych zerem. Po opis wielu z tych podprogramów sięgnij do poniższych sekcji:

- „Funkcje ciągów w Standardowej Bibliotece UCR”
- „StrBDel, StrBDelm”
- „Strcat, Strcatl, Strcatm, Strcatml”
- „Strchr”
- „Strcmp, Strcmpl, Stricmp, Stricmpl”
- „Strcpy, Strcpyl, Strdup, Strdupl”
- „Strdel ,Strdelm”
- „Strins, Strinsl, Strinsm, Strinsml”
- „Strlen”
- „Strlwr, Strlwrm, Strupr, Struprm”
- „Strrev, Strrevm”
- „Strset, Strsetm”
- „Strspan, Strspanl. Strcspan, Strcspanl”
- „Strstr, Strstrl”
- „Strtrim, Strtrimm”
- „Inne podprogramy ciągów w Bibliotece Standardowej UCR”

Jak wspomniano wcześniej, instrukcje ciągów są całkiem użyteczne dla wielu operacji poza manipulowaniem ciągiem znaków. Rozdział ten zamyka sekcje opisujące inne zastosowania dla instrukcji ciągów.

- „Zastosowanie instrukcji ciągów z innymi typami danych”
- „Ciągi całkowite o wielokrotnej precyzji”
- „Działanie z całymi tablicami i rekordami”

Zbiór jest innym powszechnym abstrakcyjnym typem danych znajdowanym w dzisiejszych programach. Zbiór jest strukturą danych, która przedstawia członków (lub brak takowych) jakiejś grupy obiektów. Jeśli wszystkie obiekty są tego samego podstawowego typu i jest ograniczona liczba możliwych obiektów w tym zbiorze, wtedy możemy użyć wektora bitów (tablicy wartości boolowskich) dla przedstawienia zbioru. Implementacja wektora bitów jest bardzo wydajny dla małych zbiorów. Biblioteka Standardowa UCR

dostarcza kilku podprogramów do manipulacji zbiorami znaków i innymi zbiorami z maksimum 256 składowymi.

- „Podprogramy zbioru znaków w Bibliotece Standardowej UCR”

15.11 PYTANIA

1) Do czego są używane przedrostki powtórzenia?
2) Jakie przedrostki ciągów są używane z następującymi instrukcjami?
   a) MOVS b) CMPS c) STOS d) SCAS
3) Dlaczego nie ma, zwykle używanych, przedrostków powtórzeń z instrukcją LODS/
4) Co się stanie z rejestrami SI,DI i CX kiedy jest wykonywana instrukcja MOVSB (bez przedrostka powtórzenia) i :
   a) jest ustawiona flaga kierunku b) flaga kierunku jest wyzerowana
5) Wyjaśnij jak działają instrukcje MOVSB i MOVSW. Opisz jak wpływają one na pamięć i rejestry z i bez przedrostka powtórzenia. Opisz co się stanie kiedy flaga kierunku jest ustawiona i wyzerowana.
6) Jak zachowamy wartość flagi kierunku przy wywołaniu procedury?
7) Jak możemy zapewnić, że flaga kierunku zawsze zawierać będzie właściwą wartość przed instrukcją ciągu bez zachowania jej wewnątrz procedury?
8) Jak jest różnica pomiędzy instrukcjami „MOVSB”, „MOVSW” i „MOVS oprnd1, oprnd2”?
9) Rozważ definicję tablicy Pascalowskiej:

```
a: array [0..31] of record
        a,b,c :char
        i,j,k: integer;
    end;
```

   Zakładając, że A[0] zostało zainicjalizowane jakąś wartością, wyjaśnij jak można użyć instrukcji MOVS do inicjalizacji pozostałych elementów A taką samą wartością jak w A[0}
10) Podaj przykład działania MOVS kiedy wymagane jest aby flaga kierunku była:
    a) wyzerowana b) ustawiona

11) Jak działa instrukcja CMPS? (Co robi? Jak wpływa na rejestry i flagi itp.)
12) Który segment zawiera ciąg źródłowy” Ciąg przeznaczenia?
13) Do czego używamy instrukcji SCAS?
14) Jak szybko można zainicjalizować całą tablicę zerami?
15) Jak są używane instrukcje LODS i STOS do zbudowania złożonych operacji ciągów
16) Jak można zastosować funkcję SUBSTR do wydobycia podciągu o długości 6 poczynając od offsetu 3 w zmiennej StrVar przechowując podciąg w zmiennej NewStr?
17) Jaki rodzaj błędu może wystąpić kiedy jest wykonywana funkcja SUBSTR?
18) Podaj przykład demonstrujący użycie każdej z poniższych funkcji ciągu:
    a) INDEX b) REPEAT c) INSERT d) DELETE e) CONCAT
19) Napisz krótką pętlę, która mnoży każdy element jednowymiarowej tablicy przez 10. Użyj instrukcji ciągów do pobrania i przechowania każdego elementu tablicy.
20) Biblioteka Standardowa UCR nie dostarcza podprogramu STRCPYM. Jaki jest podprogram, który wykonuje to zadanie?
21) Przypuśćmy, że napisałeś „grę przygodową” w której gracz wypisuje zdania a ty chcesz wybrać dwa słowa „GO” i „NORTH”, jeśli są obecne, w lini wejściowej. Jakiej (nie StdLib UCR) funkcja pojawiająca się w tym rozdziale użyłbyś do wyszukania tych słów? Jaki podprogram Biblioteki Standardowej UCR wykonałby to?
22) Wyjaśnij jak wykonać porównanie całkowite o podwyższonej precyzji używając CMPS.