lab_6 - Instrukcja do ćwiczenia

Teoria:

Celem zadania jest wczytanie danych (ciągu znaków) z pliku **stdin** (klawiatury), wyświetlenie oryginalnych danych a następnie przetworzenie danych zgodnie z pewną regułą. Znaki przetwarzane są kolejno w pętli zbudowanej w oparciu o instrukcję **LOOP** (liczba znaków jest znana z góry), a po przetworzeniu wszystkich znaków są one ponownie wyświetlane (w zmodyfikowanej formie) i program kończy swoje działanie. Przetwarzany znak znajduje się w rejestrze **%al** – instrukcja **LODSB** przenosi znak z pamięci (buforu danych) do tego rejestru, zaś zapis znaku (po ewentualnej modyfikacji) dokonywany jest przy pomocy instrukcji **STOSB**. W czasie ćwiczenia zastosowanych zostanie kilka prostych reguł: zamiana małych liter na wielkie, wielkich na małe, zmiana wielkości litery i zastępowanie cyfr określonym znakiem.

Program bazuje w znacznej części na zawartości pliku **lab_5b**, więc celowe jest wcześniejsze wykonanie ćwiczenia **lab_5**. Wykorzystanie jako źródła danych pliku **stdin** (klawiatury) w połączeniu ze zwiększeniem rozmiaru bufora na dane do **1024** bajtów, praktycznie eliminuje problem „przepełnienia" bufora (i pojawienia się komunikatu „File too big!").

Główna pętla programu ma następującą postać:

```
CLD         # (Clear Direction Flag) DF=0
next:
     LODSB      # %al = MEM[ %rsi ]; %rsi++;
                          …
          ewentualna modyfikacja %al
                          …
     STOSB      # MEM[ %rdi ] = %al; %rdi++;
     LOOP next
```

Zastosowanie instrukcji **LODSB** i **STOSB** pozwala skupić uwagę na przetwarzaniu danych poprzez ukrycie szczegółów związanych z transferem danych z i do pamięci. Należy pamiętać, aby przed pętlą nadać odpowiednie wartości rejestrom: **%rcx** (liczba znaków do przetworzenia), **%rsi** (wskaźnik (adres) źródłowy – wykorzystywany przy odczycie) oraz **%rdi** (wskaźnik (adres) docelowy – wykorzystywany przy zapisie). W programie znaki po przetworzeniu są zapisywane w tym samym miejscu gdzie były wcześniej, więc przed pętlą **%rdi** == **%rsi**.

Praktyka (lab_6.s):

Kod:

- **program główny lab_6**
  - wyświetlenie komunikatu „String:”
  - odczyt danych
  - wyświetlenie komunikatu „Before:”
  - wyświetlenie wczytanych danych
  - pętla przetwarzania danych
  - wyświetlenie komunikatu „After:”
  - wyświetlenie przetworzonych danych
  - wyświetlenie komunikatu „File too big!” (o ile faktycznie wystąpił)
  - wyświetlenie komunikatu „File error!” (o ile faktycznie wystąpił)
  - zakończenie działania programu

Testowanie:

Na etapie testowania programu (sprawdzania poprawności realizacji reguł przetwarzania) należy pamiętać o umieszczaniu wśród wprowadzanych znaków takich, które pozwolą dokładnie zweryfikować działanie. Przykładowo – jeśli modyfikowane mają być wyłącznie znaki stanowiące spójny podzbiór (przedział), to wśród danych powinny pojawić się znaki spoza przedziału, z wnętrza przedziału i wartości brzegowe (granice przedziału) – najlepiej z obu stron granicy.

Grupy znaków (podział ze względu na wartości kodów ASCII):

```
spacja
 !”#$%&'()*+,-./
0123456789
:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ
[\]^_`
abcdefghijklmnopqrstuvwxyz
{|}~
```

Działania:

1. C (Compile) – polecenie: **as –o lab_6.o lab_6.s**
2. L (Link) – polecenie: **ld –o lab_6 lab_6.o**
3. Linker sygnalizuje brak symboli „**stdin**”, „**stdout**” i „**stderr**” i „**aftflen**”.
4. Dodajemy w kodzie źródłowym definicje:

```
.equ stdin, 0
.equ stdout, 1
.equ stderr, 2
```

5. Ostatni symbol (etykietę) poprawiamy na “**aftlen**”.

6. CLR (Compile, Link, Run) – polecenie: **./lab_6**
7. Pojawiają się jednocześnie różne komunikaty oraz trochę „śmieci”. Problem polega na niewłaściwym użyciu zmiennych, które przechowują informacje o długościach poszczególnych napisów – chodzi o symbole: **promptlen**, **bufsize**, **b_read**, **beflen**, **aftlen**, **toolen** i **errlen**.

   `$variable` – adres zmiennej
   `variable` – wartość zmiennej
8. Usuwamy $ przy powyższych symbolach (głównie w instrukcjach **MOV …, %rdx**, ale nie tylko!) bo chcemy uzyskać odwołania do wartości zmiennych, a nie ich adresów.
9. CLR
10. Pojawia się komunikat „String:”.
11. Wprowadzamy dowolny łańcuch znaków – np. „Ala ma kota” – i naciskamy klawisz **Enter**.
12. Efekt działania programu powinien być mniej więcej taki:

```
buba@buba-pc:~/asm/l6$ ./lab_6
String: Ala ma kota
Before:
Ala ma kota
After:
Ala ma kota
buba@buba-pc:~/asm/l6$
```

13. Istniejące instrukcje CMP (w głównej pętli) w połączeniu z instrukcjami skoków warunkowych (JB, JA) dokonują selekcji znaków będących małymi literami. Wykorzystujemy to do zamiany małych liter na wielkie.
14. Kody małych liter są większe od kodów odpowiadających im wielkich liter o **32**, więc w miejsce instrukcji która nie dała widocznych efektów (**OR $0x20, %al**), wstawiamy instrukcję **SUB $32, %al**.
15. CLR
16. Małe litery powinny zostać zamienione na ich wielkie odpowiedniki.
17. Ponieważ we wszystkich kodach małych liter bit nr **5** (**0** – najmniej znaczący bit, **7** – najbardziej znaczący) ma wartość **1**, a w kodach wielkich liter ten sam bit ma wartość **0**, to możliwe jest użycie operacji logicznej (zmieniającej stan bitu) zamiast operacji arytmetycznej. Potrzebną operacją jest iloczyn logiczny z wartością, która zmieni bit nr **5** na **0** i nie zmieni stanu pozostałych bitów czyli **1101 1111** = **0xDF**.
18. Zastępujemy instrukcję odejmowania instrukcją **AND $0xDF, %al**.
19. CLR
20. Małe litery powinny zostać ponownie zamienione na ich wielkie odpowiedniki.
21. Aby program zamieniał wielkie litery na małe, konieczne są drobne modyfikacje kodu – zmieniamy granice wybieranego przedziału (instrukcje **CMP** – zamiast ‘**a**’ ma być ‘**A**’, zamiast ‘**z**’ ma być ‘**Z**’) a do samej zamiany używamy instrukcji dodawania: **ADD $0x20, %al.**
22. CLR
23. Wielkie litery powinny zostać zamienione na małe.

24. Podobny efekt powinniśmy uzyskać stosując operację logiczną – tym razem potrzebną operacją jest suma logiczna z wartością, która zmieni bit nr **5** na **1** i nie wpłynie na stan pozostałych bitów czyli **0010 0000** = **0x20**.
25. Zastępujemy instrukcję dodawania instrukcją **OR $0x20, %al**.
26. CLR
27. Ponownie wielkie litery powinny zostać zamienione na małe.
28. Kolejnym zadaniem jest jednoczesna zamiana małych liter na wielkie, a wielkich na małe czyli zmiana wielkości liter. Ponieważ kody wielkich liter i kody małych liter stanowią dwa rozłączne przedziały, konieczne jest zastosowanie **4** porównań. Do samej zamiany najwygodniej jest użyć operacji logicznej zmieniającej stan **5** bitu na przeciwny – taką operacją jest **XOR** z wartością, która zmieni wyłącznie stan **5** bitu czyli **0010 0000** = **0x20**.
29. Zmodyfikuj kod programu tak, aby zawartość pętli miała następującą postać:

```
        CMP  $'A', %al
        JB skip
        CMP  $'z', %al
        JA skip
        CMP  $'Z', %al
        JBE  change
        CMP  $'a', %al
        JB skip
change:
        XOR  $0x20, %al
skip: STOSB
```

30. CLR
31. Znaki będące literami powinny zmienić swoją wielkość, inne znaki nie powinny zostać zmodyfikowane.
32. Wprowadzamy do programu kolejną regułę: wszystkie cyfry mają zostać zamienione na znak '**#**'. Konieczne jest dodanie dwóch porównań i operacji zamiany.
33. Modyfikujemy kod programu tak, aby wyglądał następująco:

```
        CMP  $'0', %al
        JB skip
        CMP  $'9', %al
        JA letter
        MOV  $'#', %al
        JMP  skip
letter:
        CMP  $'A', %al
        JB skip
        …
```

    Instrukcje już istniejące zaznaczono na czerwono.

34. CLR
35. Małe litery są zamieniane na wielkie, wielkie na małe, a cyfry zastępowane są znakiem '#'.
36. Znowu sukces - brawo Ty!