Wrocław 8 marca 2015

Kamil Cała
209954
Środa 7:15 TN

Sprawozdanie z laboratorium nr 1

Data laboratorium: 04.03.2015r
Rok akademicki 2014/2015, Informatyka
Prowadzący: Mgr. Aleksandra Postawka

Opis ćwiczenia

Celem tego ćwiczenia było zapoznanie się działaniami arytmetycznymi, operacjami na ciągach znaków i tworzeniem prostych pętli w języku GNU/Assembly. Samo ćwiczenie polegało na zaimplementowaniu szyfru cezara, który polega na przesunięciu każdego znaku w zadanym ciągu o taką samą ilość pozycji w alfabecie. Przykładowo więc, jeżeli przesuwamy o 3 pozycje, litera ‘A’ zostanie zamieniona w ‘D’, a litera ‘B’ w ‘E’.

Działanie programu przebiega następująco:

1. Pobranie ciągu znaków z wejścia standardowego.
2. Zakodowanie go przy użyciu szyfru cezara. Liczba pozycji o które zostanie przesunięta każda litera jest ustalana w momencie kompilacji poprzez ustawienie wartości zmiennej SHIFT. Litery nie zmieniają swojej wielkości podczas kodowania. Znaki niebędące literami pozostają niezmienione. Jeżeli wartość litery po przesunięciu przekroczy wartość litery ‘z’, zostanie ona “zapętlona” do początku alfabetu poprzez operację modulo.
3. Wypisanie zakodowanego ciągu znaków na wyjściu standardowym.

Przebieg ćwiczenia

Wartość o którą zostanie przesunięta każda litera, oraz liczbę liter w angielskim alfabecie zapisałem w stałych:

```
.data
(...)
SHIFT = 72
ALPHABET = 26
```

Następnie zdeklarowałem dwa bufory do zapisu ciągu znaków odczytanego z wejścia standardowego, oraz do zapisu zakodowanego już ciągu. Oba mogą pomieścić 512 znaków.

```
.bss
.comm user_input, 512
.comm output, 512
```

Po pobraniu ciągu znaków z wejścia standardowego, przygotowałem wszystkie zmienne potrzebne do stworzenia pętli, a więc długość ciągu, oraz licznik.

```
movq %rax,%r10                #length of input
dec %r10                      #decrease by 1 to skip '\n'
movq $0, %r11                 #initialize counter
```

Pętla

Najważniejsza część tego programu, a mianowicie pętla iterująca po kolejnych znakach ciągu zaczyna się od zadeklarowania etykiety:

```
_caesar:
```

Natępnie, znak na który akurat wskazuje licznik pętli (zaczynając oczywiście od pierwszego), zostaje skopiowany do rejestru r12. Co warto wspomnieć, w tym własnie miejscu natknąłem się na problem przez który nie ukończyłem programu na zajęciach laboratoryjnych. Użycie instrukcji movq kopiowało z bufora do rejestru r12 jedynie 8 najmłodszych bitów które zawierały znak zakodowany w ASCII. Starsze bity rejestru nie były jednak w żaden sposób modyfikowane. Sam rejestr nie został przeze mnie wyzerowany, przez co zawierał “śmieciowe” dane. W dalszej części programu powodowało to błędy w momencie użycia instrukcji porównywającej cmp, co uniemożliwiało prawidłowe działanie programu. Problemowi temu można zaradzić poprzez:

a) Wcześniejsze wyzerowanie rejestru
b) Porównywanie jedynie ośmiu najmłodszych bitów w każdej iteracji pętli

Pierwsza metoda jest bardziej poprawna pod względem architektury całej aplikacji, a szczególnie jej dalszego utrzymania i modyfikowania. Niewyzerowane zmienne mogłyby w przyszłości spowodować trudne do wykrycia błędy. W związku z tym, instrukcję movq zastąpiłem instrukcją **movzbq** która kopiuje wartość bufora lub rejestru, wypełniając jednak pozostałe jego pozycje zerami. Składnia tej komendy jest następująca:
movz - “trzon komendy”
b - wielkość wartości którą kopiujemy. b-binary (8 bitów), w - wide (16 bitów) itd.
q - wielkośc zmiennej do której kopiujemy. W tym wypadku ‘q’, czyli 64 bity.

```
movzbq user_input(,%r11,1), %r12
```

Kolejnym krokiem było sprawdzenie czy znak znajduje się w zakresie od 65 (‘A’) do 122 (‘z’), w którym znajdują się wszystkie litery w kodowaniu ASCII. Jeżeli znak znajduje się poza tym zakresem, to można stwierdzić że na pewno nie jest literą.

```
#check if within letter-range
cmp $'A', %r12d        #if less than 'A', than definitely not a letter
jl _other
cmp $'z', %r12d        #the same if bigger than 'z'
jg _other
```

We wspomnianym zakresie znajduje się dodatkowo mniejszy zakres, pomiędzy wielkimi oraz małymi literami w którym również występują znaki inne niż litery. Należy więc porównać aktualnie przetwarzany znak z tym zakresem i na tej podstawie stwierdzić czy jest małą literą, wielką literą czy innym znakiem.

```
#mind the gab between lower and uppercase letters
cmp $'Z', %r12d       #if less/equal 'Z' - uppercase letter
jle _upper
cmp $'a', %r12d       #if greater/equal - lowercase letter
jge _lower
jmp _other            #char inside the gap
```

Jeżeli przetwarzany znak okazał się ostatecznie nie być literą, nie są przeprowadzane na nim żadne dalsze operacje i jest on bez zmian kopiowany do bufora wyjściowego.

```
_other:
 nop                   #do nothing for non-letters
```

Jeżeli jednak znak jest literą, należy go wpierw zakodować. W wypadku wielkich liter proces wygląda następująco:

```
sub $'A', %r12        #substract value of 'A' in order to normalize number to 0
add %r15, %r12 #add shift
movl %r12d, %eax    #move dividend to eax
CDQ            #/Dividend is edx:eax where edx are older bytes. This command
               #sign-extends 32bit eax to 64bit number stored in eax and edx/
div %r13               #divide the number by number of letters in the alphabet
movl %edx, %r12d       #retreive the remainder
add $'A', %r12         #take it back to place (denormalization)
```

Pierwszym krokiem jest więc odjęcie od litery wartości litery ‘A’, aby wartość znaku odpowiadała numerowi danej litery w alfabecie. Następnie dodawana jest wartość przesunięcia w celu zakodowania ciągu znaków szyfrem cezara. Operacja ta może spowodować iż wartośc znaku stanie się większa niż liczba liter w alfabecie. Aby temu zapobiec, wartość zostanie “zapętlona” poprzez wykonanie operacji modulo. W

GNU/Assembly reszta z dzielenia zapisywana jest w rejestrze edx (dla wersji 32 bitowej) przy operacji dzielenia.

Operacja dzielenia przeprowadzana jest w taki sposób, że dzielna musi znajdować się w rejestrach eax (młodsze 32 bity) i edx (starsze 32 bity). Dzielnik podawany jest w formie argumentu do instrukcji. W powyższym kodzie wartośc przetwarzanego znaku kopiowana jest więc do rejestru eax (wartość znaku ASCII z pewnością zmieści się w 32 bitach). Następnie używana jest komenda CDQ która rozszerza wartośc rejestru eax także na rejestr edx, w wypadku liczb dodatnich wypełniając go zerami, a przy liczbach ujemnych - jedynkami. Dzięki temu unikniemy problemów związanych z ewentualnymi śmieciowymi danymi które mogą znajdować sie w rejestrze edx.

Nastepnie przeprowadzana jest operacja dzielenia, gdzie dzielnikiem jest liczba znaków alfabetu zapisana w rejestrze r13. Na koniec wartosc reszty z dzielenia jest kopiowana do rejestru r12, w celu dalszego użytku. Należy dodać do niej spowrotem wartośc litery ‘A’, aby znak znalazł się w odpowiednim miejsu w tablicy ASCII.

Dla małych liter operacja wygląda analogicznie, a jedynie wartość ‘A’ jest zastąpiona przez ‘a’.

Ostatnią rzeczą którą należy wykonać w pętli jest skopiowanie znaku do buforu wyjściowego, zwiększenie licznika i sprawdzenie warunku kończącego przebieg.

```
movq %r12, output(,%r11,1)  #move the processed letter to the output buffer
inc %r11                     #increase the counter
cmp %r11, %r10        #check for loop's ending condition (every char literated)
jne _caesar                  #if not met - jump to the beginning
```

Po zakończeniu pętli do ciągu znaków z powrotem dodawany jest znak końca lini ‘\n’, a zakodowany ciąg wysyłany jest na wyjście standardowe.

Podsumowanie

Wykonanie tego ćwiczenia pozwoliło na oswojenie sie ze specyficzną formą przeprowadzania operacji arytmetycznych w Assembly. Rozwinęło też w dalszym stopniu moją umiejętność tworzenia pętli i skoków warunkowych w tym języku, a także przypomniało mi o konieczności zerowania zmiennych podczas programowania w językach niższego poziomu.