Sprawozdanie z Laboratorium Assemblera

Grzegorz Szczepanek
nr indeksu: 280678

04.04.2025

Spis treści

1 Wprowadzenie

Niniejsze sprawozdanie przedstawia realizację trzech zadań laboratoryjnych z kursu assemblera, których celem było:

- Przetworzenie ciągu znaków pobranego ze standardowego wejścia, z zamianą małych liter na wielkie, wielkich na małe oraz dodatkową inwersją co drugiej litery.
- Konwersja ciągu znaków zapisanych w systemie szesnastkowym (hex) na wartość binarną.
- Konwersja ciągu znaków reprezentujących liczbę dziesiętną na zapis binarny.

W kolejnych sekcjach opisano zaimplementowane rozwiązania wraz z komentarzami do kodu.

2 Zadanie 1: Przetwarzanie ciągu znaków

Celem zadania było pobranie ciągu znaków ze standardowego wejścia oraz zmodyfikowanie jego zawartości. Program wykonuje następujące operacje:

- Wyświetlenie komunikatu z prośbą o wprowadzenie tekstu.
- Odczytanie ciągu znaków.
- Dla każdej litery: zamiana małych liter (ASCII 97-122) na wielkie (poprzez odejmowanie 32) oraz wielkich (ASCII 65-90) na małe (poprzez dodawanie 32).
- Dodatkowo, dla liter znajdujących się na nieparzystych indeksach (np. 1, 3, 5, itd.) wykonywana jest kolejna inwersja wielkości.

Poniżej przedstawiono kod źródłowy rozwiązania:

```
.data
msg1: .ascii "Podaj␣tekst:␣"
msg1_len = . - msg1
msg2: .ascii "Przekszta cony␣tekst:␣"
msg2_len = . - msg2
buffer: .space 100
buffer_len = . - buffer

.text
.global _start

_start:
    # wy wietl 'podaj␣tekst'
    mov $4, %eax
    mov $1, %ebx
    mov $msg1, %ecx
    mov $msg1_len, %edx
    int $0x80

    # czytamy stdin
    mov $3, %eax          # setup syscalla na read syscall
    mov $0, %ebx          # file descriptor 0 (standardowe wejscie)
```

```
23      mov $buffer, %ecx      # adres w ktorym zapisujemy input
24      mov $buffer_len, %edx # maksymalna liczba bitow do odczytania
25      int $0x80              # wykonujemy syscall
26      mov %eax, %esi         # zapisujemy bajty ktore faktycznie
            odczytalismy
27
28      # Przetwarzanie inputu - zamiana wielkosci liter
29      mov $0, %edi           # inicjalizacja licznika pozycji znaku
30      mov $buffer, %ebx      # wskaznik na pocz tek buffera
31
32  process_loop:
33      cmp %esi, %edi         # sprawdzamy czy przetworzyli my wszystkie
            znaki
34      jge process_done       # jesli tak, konczymy
35
36      movb (%ebx), %cl       # pobieramy aktualny znak
37
38      # sprawdzamy czy to litera
39
40      # sprawdzamy czy to mala litera (ASCII: 97-122)
41      cmp $97, %cl
42      jl check_uppercase
43      cmp $122, %cl
44      jg check_uppercase
45
46      # to mala litera, zamieniamy na duza (odejmujemy 32)
47      subb $32, %cl
48      jmp check_second_letter
49
50  check_uppercase:
51      # sprawdzamy czy to duza litera (ASCII: 65-90)
52      cmp $65, %cl
53      jl not_letter
54      cmp $90, %cl
55      jg not_letter
56
57      # to duza litera, zamieniamy na mala (dodajemy 32)
58      addb $32, %cl
59
60  check_second_letter:
61      # sprawdzamy czy to co druga litera (indeksy 1, 3, 5, itd.)
62      mov %edi, %edx
63      and $1, %edx           # sprawdzamy czy nieparzysta pozycja
64      jz store_and_continue # jesli parzysta, pomijamy druga inwersje
65
66      # dla nieparzystych pozycji (1, 3, 5, itd.), odwracamy wielkosc
            jeszcze raz
67
68      # sprawdzamy czy teraz jest mala (ASCII: 97-122)
69      cmp $97, %cl
70      jl second_check_uppercase
71      cmp $122, %cl
72      jg second_check_uppercase
73
74      # to mala litera, zamieniamy na duza (odejmujemy 32)
75      subb $32, %cl
76      jmp store_and_continue
77
```

```
78  second_check_uppercase:
79      # sprawdzamy czy teraz jest duza (ASCII: 65-90)
80      cmp $65, %cl
81      jl store_and_continue
82      cmp $90, %cl
83      jg store_and_continue
84
85      # to duza litera, zamieniamy na mala (dodajemy 32)
86      addb $32, %cl
87      jmp store_and_continue
88
89  not_letter:
90      # brak zmian dla znakow niebedacych literami
91
92  store_and_continue:
93      movb %cl, (%ebx)      # zapisujemy zmodyfikowany znak z powrotem
94
95      inc %ebx              # przechodzimy do nastepnego znaku
96      inc %edi              # zwiekszamy licznik
97      jmp process_loop      # kontynuujemy petle
98
99  process_done:
100     # wyswietl 'przeksztalcony tekst'
101     mov $4, %eax
102     mov $1, %ebx
103     mov $msg2, %ecx
104     mov $msg2_len, %edx
105     int $0x80
106
107     # wyswietl przetworzony input
108     mov $4, %eax
109     mov $1, %ebx
110     mov $buffer, %ecx
111     mov %esi, %edx        # uzywamy liczby bitow ktore odczytalismy
112     int $0x80
113
114     mov $1, %eax
115     mov $0, %ebx
116     int $0x80
```

Listing 1: Kod źródłowy - Zadanie 1

3 Zadanie 2: Konwersja ciągu znaków hex na system binarny

W tym zadaniu użytkownik wprowadza ciąg znaków reprezentujących cyfry w systemie szesnastkowym (0-9, A-F lub a-f). Program:

- Wyświetla komunikat i pobiera ciąg znaków.
- Dla każdego znaku mnoży dotychczasowy wynik przez 16 (przesunięcie o 4 bity w lewo) i dodaje wartość cyfry, uzyskaną przez konwersję kodu ASCII.
- Po przetworzeniu wszystkich znaków konwertuje wynik (w postaci liczby dziesiętnej) na zapis binarny.

Kod źródłowy rozwiązania przedstawiono poniżej:

```
.data
msg1: .ascii "Podaj␣ciag␣znakow␣w␣hex␣(0-9,␣A-F):␣"
msg1_len = . - msg1
msg2: .ascii "Wynik␣w␣systemie␣binarnym:␣"
msg2_len = . - msg2
buffer: .space 100
buffer_len = . - buffer
result: .space 100
result_len = . - result

.text
.global _start

_start:
    # wyswietl prosbe o podanie tekstu
    mov $4, %eax
    mov $1, %ebx
    mov $msg1, %ecx
    mov $msg1_len, %edx
    int $0x80

    # czytamy stdin
    mov $3, %eax          # syscall dla read
    mov $0, %ebx          # file descriptor 0 (stdin)
    mov $buffer, %ecx     # adres bufora
    mov $buffer_len, %edx # maksymalna liczba znakow
    int $0x80

    mov %eax, %esi        # zapisujemy faktyczna dlugosc wczytanego
       tekstu
    dec %esi              # pomijamy znak nowej linii

    # przygotowujemy rejestry
    mov $0, %edi          # indeks znaku
    mov $buffer, %ebx     # wskaznik na bufor
    mov $0, %eax          # czyszczenie eax - tu bedziemy trzymac wynik

process_loop:
    cmp %esi, %edi        # sprawdzamy czy przetworzyli my wszystkie
       znaki
    jge output_binary     # jesli tak, konczymy przetwarzanie

    movb (%ebx), %cl      # pobieramy aktualny znak

    # mnozymy aktualny wynik przez 16 (przesuniecie o 4 bity w lewo)
    shl $4, %eax

    # sprawdzamy jakiego typu jest znak
    cmp $'0', %cl
    jl process_error      # jesli < '0', blad
    cmp $'9', %cl
    jle digit_0_to_9      # jesli <= '9', to cyfra 0-9

    cmp $'A', %cl
    jl process_error      # jesli < 'A', blad
    cmp $'F', %cl
    jle letter_A_to_F     # jesli <= 'F', to litera A-F
```

```
56
57      cmp $'a', %cl
58      jl process_error        # jesli < 'a', blad
59      cmp $'f', %cl
60      jle letter_a_to_f       # jesli <= 'f', to litera a-f
61
62      jmp process_error       # jesli inny znak, blad
63
64  digit_0_to_9:
65      subb $'0', %cl          # konwertujemy ASCII na wartosc numeryczna
66      jmp add_digit
67
68  letter_A_to_F:
69      subb $'A', %cl          # odejmujemy kod ASCII dla 'A'
70      addb $10, %cl           # dodajemy 10, bo 'A' = 10 w hex
71      jmp add_digit
72
73  letter_a_to_f:
74      subb $'a', %cl          # odejmujemy kod ASCII dla 'a'
75      addb $10, %cl           # dodajemy 10, bo 'a' = 10 w hex
76
77  add_digit:
78      add %cl, %al            # dodajemy wartosc do wyniku
79
80  next_char:
81      inc %ebx                # przechodzimy do nastepnego znaku
82      inc %edi                # zwiekszamy licznik
83      jmp process_loop
84
85  process_error:
86      # tu mozna dodac obsluge bledu
87      jmp next_char
88
89  output_binary:
90      # eax zawiera teraz liczbe w systemie dziesietnym
91      # konwertujemy na reprezentacje binarna
92      mov $result, %ecx       # adres bufora wynikowego
93      mov $32, %edx           # liczba bitow (dla 32-bitowej liczby)
94      add %edx, %ecx          # idziemy na koniec bufora
95      movb $0, (%ecx)         # dodajemy null terminator
96      dec %ecx                # cofamy sie o jeden znak
97
98  convert_binary_loop:
99      movb $'0', (%ecx)       # domyslnie wstawiamy '0'
100     test $1, %eax           # sprawdzamy najmlodszy bit
101     jz skip_one_bit
102     movb $'1', (%ecx)       # jesli bit = 1, wstawiamy '1'
103
104 skip_one_bit:
105     shr $1, %eax            # przesuwamy eax w prawo o 1 bit
106     dec %ecx                # cofamy wskaznik bufora
107     dec %edx                # zmniejszamy licznik bitow
108     jnz convert_binary_loop
109
110     inc %ecx                # przesuwamy wskaznik na pierwszy znaczacy bit
111
112     # wyswietl komunikat
113     mov $4, %eax
```

```
114     mov $1, %ebx
115     mov $msg2, %ecx
116     mov $msg2_len, %edx
117     int $0x80
118
119     # oblicz dlugosc wyniku binarnego
120     mov $result, %edx
121     add $32, %edx          # przechodzimy na koniec bufora
122     sub %ecx, %edx         # obliczamy dlugosc
123
124     # wyswietl wynik binarny
125     mov $4, %eax
126     mov $1, %ebx
127     # ecx juz wskazuje na pierwszy znaczacy bit
128     # edx juz zawiera dlugosc
129     int $0x80
130
131     # koniec programu
132     mov $1, %eax
133     mov $0, %ebx
134     int $0x80
```

Listing 2: Kod źródłowy - Zadanie 2

4 Zadanie 5: Konwersja ciągu znaków dziesiętnych na system binarny

W zadaniu 5 ciąg znaków reprezentujący liczbę dziesiętną jest przetwarzany w następujący sposób:

- Program pobiera ciąg znaków (cyfry 0-9) ze standardowego wejścia.
- Dla każdej cyfry aktualny wynik mnożony jest przez 10 (realizowane poprzez przesunięcia: `shl $3` oraz `shl $1` i dodawanie), a następnie dodawana jest wartość cyfry.
- Po przetworzeniu całego ciągu liczba konwertowana jest na postać binarną.

Poniżej przedstawiono kod źródłowy:

```
1  .data
2  msg1: .ascii "Podaj␣ciag␣znakow␣w␣systemie␣dziesietnym␣(0-9):␣"
3  msg1_len = . - msg1
4  msg2: .ascii "Wynik␣w␣systemie␣binarnym:␣"
5  msg2_len = . - msg2
6  buffer: .space 100
7  buffer_len = . - buffer
8  result: .space 100
9  result_len = . - result
10
11 .text
12 .global _start
13
14 _start:
15     mov $4, %eax
16     mov $1, %ebx
17     mov $msg1, %ecx
```

```
18      mov $msg1_len, %edx
19      int $0x80
20
21      mov $3, %eax
22      mov $0, %ebx
23      mov $buffer, %ecx
24      mov $buffer_len, %edx
25      int $0x80
26
27      mov %eax, %esi
28      dec %esi
29
30      mov $0, %edi
31      mov $buffer, %ebx
32      mov $0, %eax
33
34  process_loop:
35      cmp %esi, %edi
36      jge output_binary
37
38      movb (%ebx), %cl
39
40      mov %eax, %edx
41      shl $3, %eax
42      shl $1, %edx
43      add %edx, %eax
44
45      cmp $'0', %cl
46      jl process_error
47      cmp $'9', %cl
48      jg process_error
49
50      subb $'0', %cl
51      add %cl, %al
52
53  next_char:
54      inc %ebx
55      inc %edi
56      jmp process_loop
57
58  process_error:
59      jmp next_char
60
61  output_binary:
62      mov $result, %ecx
63      mov $32, %edx
64      add %edx, %ecx
65      movb $0, (%ecx)
66      dec %ecx
67
68  convert_binary_loop:
69      movb $'0', (%ecx)
70      test $1, %eax
71      jz skip_one_bit
72      movb $'1', (%ecx)
73
74  skip_one_bit:
75      shr $1, %eax
```

```
76      dec %ecx
77      dec %edx
78      jnz convert_binary_loop
79
80      inc %ecx
81
82      mov $4, %eax
83      mov $1, %ebx
84      mov $msg2, %ecx
85      mov $msg2_len, %edx
86      int $0x80
87
88      mov $result, %edx
89      add $32, %edx
90      sub %ecx, %edx
91
92      mov $4, %eax
93      mov $1, %ebx
94      int $0x80
95
96      mov $1, %eax
97      mov $0, %ebx
98      int $0x80
```

Listing 3: Kod źródłowy - Zadanie 5

5 Podsumowanie

Realizacja powyższych zadań umożliwiła:

- Praktyczne zapoznanie się z operacjami na rejestrach oraz manipulacją bitową w asemblerze.
- Zrozumienie mechanizmów przetwarzania ciągów znaków oraz konwersji danych między systemami liczbowymi (hex, dziesiętnym i binarnym).
- Utrwalenie zasad wywołań systemowych w systemie Linux na architekturze x86.

6 Wnioski

Ćwiczenia pozwoliły na pogłębienie wiedzy z zakresu programowania niskopoziomowego oraz praktyczne zastosowanie operacji bitowych i konwersji danych. Prezentowane rozwiązania pokazują różne podejścia do przetwarzania danych, co stanowi istotny element w projektowaniu optymalnych aplikacji systemowych.