Wrocław, dn. 23 marca 2017

Hanna Grodzicka, 226154

Utrwalenie umiejętności tworzenia prostych konstrukcji programowych
sprawozdanie z laboratorium przedmiotu „Architektura Komputerów 2”
Rok akad. 2016/2017, kierunek: INF

Prowadzący:
mgr inż. Aleksandra Postawka

1. Cel ćwiczenia

Celem ćwiczenia były:

- odczyt z pliku dwóch dużych liczb danych w reprezentacji czwórkowej (dane w kodzie ascii);
- zamiana obu liczb na system szesnastkowy i ich zapis w pamięci;
- dodanie tych liczb z wykorzystaniem flagi CF (ang. *carry flag*) i rejestrów 64-bitowych;
- zapis wyniku do pliku w postaci liczby o podstawie systemu szesnastkowego z koniecznością zamiany liczb na odpowiedniki w ascii.

2. Przebieg ćwiczenia

W sekcji `.data` oprócz przypisania numerów dla `SYSREAD`, `SYSWRITE`, `SYSEXIT` i `EXIT_SUCCESS`, deklarowane są jeszcze numery do wywołania dwóch funkcji systemowych z systemie 64-bitowym:

```
.data
SYSOPEN      = 2
SYSCLOSE     = 3
```

Pierwsza stała będzie przechowywała numer funkcji systemowej do otwierania pliku, druga zaś, do jego zamykania.

```
O_RDONLY     = 0
O_CREAT_WRONLY_TRUNC = 03101
```

Powyższe wartości będą używane jako drugi argument wywoływanych funkcji zapisu/ odczytu z pliku. Ich numery oznaczają intencje dokonywanych zmian w pliku: dla `0` jest to czytanie, a dla `03101` pisanie do pliku.

```
END_OF_FILE  = 0
```

Po deklaracji numerów strumieni wejściowych i wyjściowych (`STDIN`, `STDOUT`), tworzona jest stała pomocnicza `END_OF_FILE` przechowująca znak, który jest zwracany przy czytaniu z pliku, jeśli osiągnięty zostaje jego koniec.

```
BUF_LEN = 512
```

Długość używanych buforów to 512 bajtów; dla wyniku i tekstu do wypisania o jeden bajt więcej, bo podczas dodawnia może wystąpić przeniesienie na najstarszą pozycję.

```
.bss
.comm   textin,  512        #bufor do pobrania tekstu z pliku
.comm   tmp,     512        #do odsiania liczb w systemie
                            #czwórkowym
.comm   liczba1, 512        #na pierwszą liczbę zapisaną liczbowo
.comm   liczba2, 512        #na drugą liczbę
.comm   wynik,   513        #na wynik dodawania liczba1 i liczba2
.comm   textout, 513        #wypisanie wyniku w ascii

.text
#pozycje na stosie
ST_SIZE_RESERVE = 16 #rezerwacja miejsca na FD plików
ST_FD_IN     = -8   #file descriptor (plik wejściowy - input)
ST_FD_OUT    = -16  #file descriptor (plik wyjściowy - output)
ST_ARGC      = 0    #liczba argumentów (3)
ST_ARGV_0    = 8    #nazwa pliku (argument 0.)
```

```
ST_ARGV_1    = 16   #plik wejściowy (argument 1.)
ST_ARGV_2    = 24   #plik wyjściowy (argument 2.)
```

Powyższe zmienne są rezerwacją miejsca na stosie dla poszczególnych wartości, których nie chcemy stracić po wywołaniu funkcji systemowych, a które będą używane w programie.
Nazwy rozpoczynające się od `ST_ARG` będą służyły jako dodany offset do wskaźnika stosu, aby umożliwić pobranie argumentów od użytkownika podczas wywoływania programu.

```
.globl _start
_start:
movq    %rsp, %rbp     #%rbp wskazuje na miejsce 0 na stosie
subq    $ST_SIZE_RESERVE, %rsp   #%rsp wskazuje na -16
```

Na chwilę obecną stos wygląda następująco:

```
#ST_ARGV_2     <--- 24(%rbp)
#ST_ARGV_1     <--- 16(%rbp)
#ST_ARGV_0     <--- 8(%rbp)
#ST_ARGC       <--- (%rbp)
#ST_FD_IN      <--- -8(%rbp)
#ST_FD_OUT     <--- -16(%rbp) i (%rsp)
```

Ostatni element jest szczytem stosu.

```
#otwarcie pliku (INPUT)
movq    $SYSOPEN, %rax
movq    ST_ARGV_1(%rbp), %rdi
movq    $O_RDONLY, %rsi
movq    $0666, %rdx
syscall
```

Przy wywołaniu funkcji do otwierania pliku w argumenty należy podać:
- w `%rdi` wskaźnik na pierwszy znak nazwy pliku (w tej wersji ten argument będzie pobrany ze stosu);

- `%rsi` zawiera intencje odczytu lub zapisu do pliku;
- `%rdx` przechowuje uprawnienia dla użytkowników (właściela, grup i innych).

Korzystając z terminala można wyświetlić te uprawnienia (i ich zmiany) za pomocą `ls -l plik`. W powyższych liniach jako numery uprawnień dla każdego wpisane jest `6`, co oznacza, że każdy ma prawo do czytania i wpisywania do pliku (`4 (read) + 2 (write) = 6`). Po wyświetleniu uprawnień w terminalu powinno się pokazać `rw-` dla uprawnienia `6`.

```
#file descriptor zwracany w %rax
movq    %rax, ST_FD_IN(%rbp)      #zapisanie FD_IN na stosie

#otwarcie pliku (OUTPUT)
movq    $SYSOPEN, %rax
movq    ST_ARGV_2(%rbp), %rdi
movq    $O_CREAT_WRONLY_TRUNC, %rsi
movq    $0666, %rdx
syscall

movq    %rax, ST_FD_OUT(%rbp)     #zapisanie FD_OUT na stosie
```

Po wywołaniu funkcji systemowej otwierania pliku, w `%rax` zwracany jest numer zwany *file descriptor* (FD), dzięki któremu można się odnosić do pliku modyfikowanego w dalszej części programu. Aby uniknąć nadpisania tych numerów, można je odłożyć na stos i zapamiętać ich pozycje, tak jak w tym programie.

```
read_loop:
movq    $SYSREAD, %rax
movq    ST_FD_IN(%rbp), %rdi      #strumień wejściowy to FD pliku
input
movq    $textin, %rsi
movq    $BUF_LEN, %rdx
syscall
#w %rax zwracana jest liczba wczytanych znaków
```

Czytanie z pliku odbywa się podobnie do czytania z klawiatury przy pomocy standardowego wejścia, tylko zamiast `$STDIN` podawany jest *file descriptor* pliku, z którego chcemy czytać. Gdy wczytanie z pliku się nie powiedzie, w `%rax` zwracana jest liczba ujemna.

```
#sprawdzanie czy udał się odczyt pliku
cmpq    $END_OF_FILE, %rax
jle     end_loop    #skok jeśli nastąpił błąd odczytu, czyli
                    #liczba ujemna w %rax lub koniec pliku ($0)
```

A. Odczyt zawartości `textin` i przepisanie jej bajt po bajcie do bufora `tmp` filtrując jednocześnie wszystkie znaki, które nie są 0, 1, 2, 3 lub spacją, która służy do oddzielenia dwóch liczb. Przed wpisaniem do `tmp` od tych cyfr odejmowana jest wartość ascii dla `$'0'`; spacja przepisywana jest normalnie. Wszystkie inne znaki nie zostają przepisane.

```
movq    $0, %rdi        #licznik dla textin
movq    $0, %r8         #licznik do pierwszej liczby
movq    $0, %r9         #licznik dla tmp
movq    $0, %rcx        #rejestr do pobierania znaków
movq    $0, %r12        #licznik do drugiej liczby
movq    $0, %r11        #'0' dla wczytywania do liczba1
                        #'1' do liczba2

read_textin:
cmpq    %rax, %rdi
jge     continue
movb    textin(,%rdi,1), %cl
incq    %rdi

cmp     $' ', %cl        #spacja w ascii to $32
je      save_nl
```

```
cmp     $'0', %cl
jl      read_textin
cmp     $'3', %cl
jg      read_textin     #pominięcie wszystkich znaków, które
                        #nie są w [0;3] za wyjątkiem ' '

subb    $'0', %cl
save_nl:
movb    %cl, tmp(,%r9,1)
incq    %r9
jmp     read_textin
```

B. Cyfry z bufora `tmp` są czytane od ostatniego indeksu i zapisywane w takiej odwrotnej kolejności do bufora `liczba2`. Po napotkaniu znaku spacji podczas odczytywania kolejnych bajtów, zaczyna się zapis do bufora `liczba1`.
Przy odczycie z `tmp` na każdym bajcie zapisane są 2 bity informacji o przechowywanej liczbie. W poniższym kodzie wykorzystane są przesunięcia bitowe, tak by każdy bajt zawierał w sobie informację o czterech liczbach.
Dzięki temu miejsce zostaje efektywnie wykorzystane nie pozostawiając „pustych” bitów, które nie wnoszą informacji. Jest to przydatne też przy późniejszej operacji dodawania, żeby móc skorzystać z flag przeniesienia.

```
continue:
dec     %r9     #cofnięcie ostatniej inkrementacji, by trafić
                #na indeks ostatnio wpisanego znaku do $tmp
go_on:
cmpq    $0, %r9
jl      compute
movq    $0, %rdx    #rejestr do pomocy przy przesunięciach

pack_2bits:
movq    $0, %rcx
movb    tmp(,%r9,1), %dl
cmp     $' ', %dl
```

```
je      switch_numbers
addb    %dl, %cl
decq    %r9
cmpq    $0, %r9
jl      to_textout

pack_4bits:
movb    tmp(,%r9,1), %dl
cmp     $' ', %dl
je      switch_numbers
shlb    $2, %dl
addb    %dl, %cl
decq    %r9
cmpq    $0, %r9
jl      to_textout

pack_6bits:
movb    tmp(,%r9,1), %dl
cmp     $' ', %dl
je      switch_numbers
shlb    $4, %dl
addb    %dl, %cl
decq    %r9
cmpq    $0, %r9
jl      to_textout

pack_8bits:
movb    tmp(,%r9,1), %dl
cmp     $' ', %dl
je      switch_numbers
shlb    $6, %dl
addb    %dl, %cl
decq    %r9
jmp     to_textout
```

```
switch_numbers:
movq    $1, %r11
decq    %r9
jmp     go_on

to_textout:
cmp     $1, %r11
je      liczb1

liczb2:
movb    %cl, liczba2(,%r12,1)
incq    %r12
jmp     go_on

liczb1:
movb    %cl, liczba1(,%r8,1)
incq    %r8
jmp     go_on
```

C. Od tego miejsca zaczyna się operacja dodawania liczb z buforów `liczba1` i `liczba2`. Obie liczby są wpisane w buforach od najmłodszego bajtu do najstarszego. Wynik dodawnia wraz z przeniesieniem jest wpisywany do bufora `wynik`.

```
compute:
xor     %rax, %rax   #do pobierania bajtów z liczba2
xor     %rbx, %rbx   #do pobierania bajtów z liczba1
xor     %rdx, %rdx   #do przechowywania flagi przeniesienia
xor     %rsi, %rsi   #licznik

cmp     %r12, %r8
jl      iterate_by_r12
iterate_by_r8:
```

```
movq    %r8, %rcx
jmp     begin
iterate_by_r12:
movq    %r12, %rcx   #iteracja po dłuższej liczbie

begin:
clc                  #zerowanie CF (carry flag)

add_loop:
movb    liczba2(,%rsi,1), %al
movb    liczba1(,%rsi,1), %bl

cmpb    $0, %dl      #sprawdzenie zachowanej CF
jne     cf_on
clc          #if(dl == 0) {flagi nie ustawiamy (zerujemy)}
jmp     add_continue

cf_on:
stc                   #ustawienie CF

add_continue:
adc     %bl, %al   #dodawanie %bl do %al włącznie z flagą CF
movb    %al, wynik(,%rsi,1)
setc    %dl        #zapisanie CF z ostatniego dodawania w %dl
inc     %rsi
loop    add_loop

movb    %dl, wynik(,%rsi,1) #dodanie przeniesienia do wyniku
```

Instrukcja loop zawiera w sobie trzy operacje: dekrementację rejestru `rcx` (`decq %rcx`), porównanie do z zerem (`cmpq $0, %rcx`) i skok, jeśli w `rcx` nie ma zera (`jne add_loop`).

D. Przepisanie wyniku dodawania z bufora `wynik`, w którym wynik operacji jest zapisany w konwencji little endian do bufora `textout`. Ta operacja obejmuje odczyt bajtów z wynik od ostatniego indeksu do pierwszego, podzielenie bajtu informacji na dwie części, tak by mieć dwie liczby heksadecymalne i dodanie odpowiedniej wartości, by uzyskać odpowiedni kod ascii dla cyfry bądź litery.

```
xor     %r15, %r15      #licznik do textout

hex_loop:
xor     %rcx, %rcx
xor     %rdx, %rdx
cmp     $0, %rsi
jl      end_loop
movb    wynik(,%rsi,1), %dl
movb    %dl, %cl
shrb    $4, %dl
andb    $0xf, %dl       #4 górne bity
andb    $0xf, %cl       #4 dolne bity
decq    %rsi

cmp     $9, %cl
jg      cl_to_letter

cl_to_number:
addb    $'0', %cl
jmp     dl_to_ascii

cl_to_letter:
addb    $55, %cl        #konwersja [10;15] na ascii w [A;F]
jmp     dl_to_ascii

dl_to_ascii:
cmp     $9, %dl
jg      dl_to_letter
```

```
dl_to_number:
addb    $'0', %dl
jmp     go_textout

dl_to_letter:
addb    $55, %dl        #konwersja [10;15] na ascii w [A;F]
jmp     go_textout

go_textout:
movb    %dl, textout(,%r15,1)
incq    %r15
movb    %cl, textout(,%r15,1)
incq    %r15
jmp     hex_loop
```

E. Zapis `textout` do pliku wyjściowego. W ostatnim argumencie funkcji podana jest liczba bajtów do wypisania. Następnie zamknięcie obu plików (wejściowego i wyjściowego) oraz wyjście z programu.

```
#zapis bufora do pliku OUTPUT
movq    %r15, %rdx
movq    $SYSWRITE, %rax
movq    ST_FD_OUT(%rbp), %rdi
movq    $textout, %rsi
syscall

jmp     read_loop     #pobiera kolejny segment tekstu

end_loop:
#zamknięcie pliku OUTPUT
movq    $SYSCLOSE, %rax
movq    ST_FD_OUT(%rbp), %rdi
```

```
    syscall

    #zamknięcie pliku INPUT
    movq    $SYSCLOSE, %rax
    movq    ST_FD_IN(%rbp), %rdi
    syscall

    #wyjście z programu
    movq    $SYSEXIT, %rax
    movq    $EXIT_SUCCESS, %rdi
    syscall
```

3. Podsumowanie i wnioski

   Niestety zadanie nie jest wykonane dokładnie z poleceniem. Dodawanie zostało zrealizowane za pomocą rejestrów ośmiobitowych, a nie 64b. Wynika to z mojej nieuwagi. Napisawszy poprzednie części zadania i widząc, że moje liczniki buforów `liczba1` i `liczba2` dotyczą bajtów, uznałam, że najprościej się trzymać tej wielkości i dodawać kolejne kawałki (bajty) liczb ze sobą, jednocześnie dekrementując licznik pobrany od dłuższej z liczb.

   W pamięci nie ma zapisanych obu liczb heksadecymalnie. Na początkowym etapie konstrukcji programu ta część była obecna. Przy późniejszej zmianie konwencji uznałam, że do dodawania przydatne będą „zbite” liczby (bez pustych bitów), żeby móc ustawić flagę przeniesienia. W rezultacie zapomniałam o tym wymogu.

   W podpunkcie C. przy dodawniu dwóch liczb, iteracja odbywa się po najdłuższej liczbie. W praktyce program wymaga podania dwóch liczb o tej samej długości - w innym wypadku nie będzie możliwe pobranie bajtu o wyższym indeksie. Rozwiązaniem problemu mogłoby być wsadzenie zer do jednego z buforów, tak by `liczba1` i `liczba2` miały takie same długości w bajtach.

4. Bibliografia

- J.Barlett *Programming from the ground up*, 2003