Wrocław, dn. 14 maja 2016r

Rafał Pieniążek, 209847
PN-P-8

prowadzący: prof. Janusz Biernat

Laboratorium Architektury Komputerów
(1)Tworzenie prostych konstrukcji programowych z użyciem instrukcji asemblera Linux/x86

1.1 Treść ćwiczenia

Zakres ćwiczenia:

- Nabycie umiejętności związanych z tworzeniem prostych pętli oraz instrukcji warunkowych z użyciem assemblera.

1.2 Przebieg ćwiczenia

1.2.1 Zamiana wielkości liter

Na początku laboratoriów został przygotowany prosty program przetwarzający wprowadzone dane. Program wczytywał dane wprowadzane z klawiatury, zamieniał wszystkie litery na małe. Program był zabezpieczony przed wprowadzaniem niepoprawnych danych. Ciąg znaków z bufora był przetwarzany do momentu napotkania znaku nowej linii.

```
SYSCALL32 = 0x80

EXIT = 1
WRITE = 4
READ = 3
STDOUT = 1
MASK = 0x20  #maska zamieniajaca kazda litere na wielka 0010 0000

.data
bufor: .space BUFOR_SIZE #
bufor_len = . - bufor

.text
.global _start

_start:

 movl $bufor_len, %edx #wczytanie
 movl $bufor, %ecx
 movl $STDOUT, %ebx
```

```
 movl $READ, %eax

 int $SYSCALL32
 #początek przetwarzenia
 xorl %edi, %edi  #inicjalizacja wskaznika

loop:

 movb bufor(,%edi,1), %al #skopiowanie  znaku z bufora do rejestru

 cmpb $'\n',%al  #sprawdzamy, czy przetworzono juz cala linie
 je print

 orb $MASK, %al  #zamiana wszystkich liter na male

 cmpb $'a', %al  #sprawdzenie, czy znak jest literą
 jl break
 cmpb $'z', %al
 jg break

 movb %al,bufor(,%edi,1)#skopiowanie znaku do bufora

 break:
 incl %edi #inkrementacja wskaznika
 jmp loop

 print: #wyswietlenie
 movl $bufor_len, %edx
 movl $bufor, %ecx
 movl $STDOUT, %ebx
 movl $WRITE, %eax

 int $SYSCALL32

 movl $EXIT, %eax
 int $SYSCALL32
```

Program został skompilowany i uruchomiony przy pomocy poniższego polecenia. Zaobserwowano poprawne działanie.

```
as cw1.s
```

1.2.2 Szyfr Cezara

Kolejnym zadaniem było zaimplementowanie algorytmu szyfrowania szyfrem Cezara. Kluczem był pierwszy znak wczytany z klawiatury. Jeżeli wczytany klucz był wielką literą, algorytm szyfrował napis, w przeciwnym przypadku następowało deszyfrowanie.

```
SYSCALL32 = 0x80

EXIT = 1
WRITE = 4
READ = 3
STDOUT = 1
MASK = 0x20 #maska zamieniajaca rozmiar liter

.data
bufor: .space BUFOR_SIZE
bufor_len = . - bufor

.text
.global _start
#male deszyfrowanie
#wielkie szyfrowanie

_start:

#wczytanie
movl $bufor_len, %edx
movl $bufor, %ecx
movl $STDOUT, %ebx
movl $READ, %eax

int $SYSCALL32

xorl %edi, %edi  #inicjalizacja wskaznika
movb bufor(,%edi,1), %bl  #wczytanie parametru szyfru
incl %edi

or $0x40, %bl  #duze kody mają kody 0x41,0x42....
cmpb $'Z',%bl #jezeli wielka to szyfrujemy
jbe prepare_encrypt
prepare_decrypt: #w przeciwnym przypadku deszyfrujemy
subb $'a', %bl
negb %bl
jmp cipher


prepare_encrypt:
subb $'A', %bl
#movb $' ',bufor(,$0,1)  #ukrycie klucza

cipher:
movb bufor(,%edi,1), %al #pobranie znaku z bufora
cmpb $'\n',%al #sprawdzenie, czy nie nastąpił znak końca nowej linii
je out
```

```
orb $MASK, %al  #zamiana wszystkich liter na male
cmpb $'a', %al  #sprawdzenie, czy znak jest literą
jl hop
cmpb $'z', %al
jg hop

addb %bl, %al #szyfrowanie

cmpb $'z',%al   #koreka szyfru
jle hop
subb $'z', %al
addb $'a', %al

hop:
movb %al,bufor(,%edi,1) #wpisanie przetworzonego napisu do bufora
incl %edi #zwiększenie wskaźnika
jmp cipher

out:
movl $bufor_len, %edx #wyswietlenie
movl $bufor, %ecx
movl $STDOUT, %ebx
movl $WRITE, %eax

int $SYSCALL32

movl $EXIT, %eax
int $SYSCALL32
```

Program wczytuje pierwszy znak i rozpoznaje jej rozmiar. Na tej podstawie obliczony jest klucz, który jest ujemny dla deszyfrowania i dodatni dla szyfrowania. Następnie algorytm przetwarza kolejne znaki wczytane z bufora. Po skopiowaniu pojedynczej litery dodawany jest do niej klucz. Następnie szyfr jest korygowany. Jeżeli po dodaniu, lub odjęciu klucza wartość znajdzie się poza zakresem alfabetu następuje odpowiednia poprawka. Program został uruchomiony przy pomocy poniższych poleceń. Po przetestowaniu stwierdzono poprawne działanie.

```
as cezar.s -o cezar.o
ld cezar.o -o cezar
./cezar
```

1.2.3 Konwersje liczbowe danych wejściowych

Kolejnym etapem było przygotowanie programów wczytujących napisy w postaci cyfr i zamieniających je na liczby. Przygotowano uniwersalną funkcję wczytującą ciąg cyfr do bufora,zamieniającą ją na liczbę o podstawie danej stałą zdefiniowaną w programie. W celu zachowania czytelności pominięto powtarzające się elementy programu.

Funkcja jest w stanie dekodować liczby zapisane w systemie o podstawie maksymalnie równej 16. Algorytm jest oparty na schemacie Hornera.

```
...
BASE = 16

.text
.global _start
_start:

#wczytanie
...

mov $0,%edx #wynik
mov $0,%edi #wskaznik
mov $0,%ecx #rejestr do przechowywania aktualnej cyfry

L1:
movb bufor(,%edi,1), %cl

cmpb $'\n',%cl #sprawdzenie, czy nie nastąpił koniec bufora
jne L2
ret
L2:
mov $BASE,%eax #mnozenie kolejnych cyfr
mul %edx
mov %eax,%edx
subb $'0',%cl
add %ecx,%edx

orb $MASK,%cl #zamiana liter na małe
cmp $'a',%cl #sprawdzenie, czy przetwarzana jest litera a-f
jb is_digit
cmp $'f',%cl
ja is_digit
subb $'a',%cl #odjęcie w celu uzyskania cyfry jednosci
add $10, %cl  #dodanie w celu uzyskani cyfry dziesiętnej
jmp hop

is_digit:
subb $'0', %cl #jezeli przetwarzana cyfra, odejmuje kod znaku '0'

hop:
add %ecx,%edx

inc %edi
jmp L1
```

1.2.4 Konwersja liczbowa danych wyjściowych

Poniżej przedstawiono fragment programu przetwarzającego liczbę przechowywaną w rejestrze *%eax* na ciąg znaków. Funkcja jest uniwersalna i może wypisywać znaki w systemie o dowolnej podstawie. Algorytm dzieli liczbę przez bazę systemu liczbowego, następnie resztę zamienia na znak danej cyfry i wynik zapisuje w buforze wyjściowym.

```
processOutputNumber:

mov $BASE_OUT,%ebx #podstawa systemu zapisana w programie
mov $outputBuforLen-2, %ecx
L3:
mov $0, %edx #wyzerowanie rejestru danej cyfry
div %ebx #dzielenie pozostałej liczby
addb $'0', %dl #zamiana cyfry na kod znaku a ASCII(reszta z dzielenia)
movb %dl,outputBufor(,%ecx,1)#skopiowanie do bufora
dec %ecx #zmniejszenie wskaznika
cmp $0,%eax #sprawdzenie warunku zakończenia algorytmu
jne L3
ret
```

Powyższe programy były kompilowane i uruchomiane następującymi poleceniami:

```
as z1.s -o z1.o
ld z1.o -o z1
./z1
```

1.3 Wnioski

Ćwiczenie pozwoliło na zaznajomienie z instrukcjami i składnią asemblera. Początkowo problematyczne było odwoływanie się do bufora, często spotykałem się z błędem *Segmentation fault*.

(2)Utrwalenie wiadomości, użycie stosu oraz innych instrukcji w języku Assembler na platformie Linux

2.1 Treść ćwiczenia

Zakres ćwiczenia:

- Tworzenie programów wykorzystujących funkcje. Konstrukcja algorytmów rekurencyjnych.

2.2 Przebieg ćwiczenia

2.2.1 Funkcja obliczeniowa

Pierwszym zadaniem było zaimplementowanie prostej funkcji obliczeniowej. W tym celu wykorzystano przygotowane na poprzednim laboratorium fragmenty kodu odpowiedzialne za wczytywanie i wypisywanie liczb. Funkcja *multiplyTwoNumbers* pobiera argumenty ze stosu, mnoży przez siebie, a wynik również odkłada na stosie.

```
SYSCALL32 = 0x80

EXIT = 1
WRITE = 4
READ = 3
STDOUT = 1
BUFOR_SIZE = 10
BASE = 10
.data
bufor: .space BUFOR_SIZE
bufor_len = . - bufor

outputBufor: .ascii "wynik:         \n"
outputBuforLen=.-outputBufor
.text
.global _start

_start:

call processInputNumber #wczytaj i przetworz pierwsza liczbe
push %rdx #odłóż wczytaną liczbę na stosie
call processInputNumber
push %rdx
call multiplyTwoNumbers #wywołanie funckji mnozacej dwie liczby
pop %rbx
mov %rbx,%rax
call processOutputNumber
```

```
jmp out

multiplyTwoNumbers:

push %rbp #
mov %rsp,%rbp

mov 16(%rbp),%rax #pobranie pierwszej liczby
mov 24(%rbp),%rdx #pobranie drugiej liczby

  mul %rdx #pomnozenie - wynik w rax

mov %rax,16(%rbp) #zapisanie wyniku dzialania funkcji
mov %rbp,%rsp  #przywrocenie wskaznika stosu szczytu
pop %rbp

ret

processInputNumber: #wczyta i przetworzy napis podany na wejsciu na liczbe dziesiet

#wczytanie
movl $bufor_len, %edx
movl $bufor, %ecx
movl $STDOUT, %ebx
movl $READ, %eax

int $SYSCALL32


mov $0,%edx #wynik
mov $0,%edi
mov $0,%ecx

L1:
movb bufor(,%edi,1), %cl

cmpb $'\n',%cl
jne L2
break1: ret
L2:
mov $BASE,%eax
mul %edx
mov %eax,%edx
subb $'0',%cl
add %ecx,%edx

inc %edi
jmp L1
```

```
processOutputNumber: #przetworzy dane z eax na ciag łańcuchów ascii i wynik przecho

mov $BASE,%ebx
mov $outputBuforLen-2, %ecx
L3:
mov $0, %edx
div %ebx
addb $'0', %dl
movb %dl,outputBufor(,%ecx,1)
dec %ecx
cmp $0,%eax
jne L3
ret

out:
#wyswietlenie
movl $outputBuforLen, %edx
movl $outputBufor, %ecx
movl $STDOUT, %ebx
movl $WRITE, %eax

int $SYSCALL32

movl $EXIT, %eax
int $SYSCALL32
```

Powyższy program został skompilowany i uruchomiony następującymi poleceniami:

```
as z1.s -o z1.o
ld z1.o -o z1
./z1
```

2.2.2 Obliczanie silni

W celu zachowania czytelności pominięto pewne fragmenty, które zostały przedstawione wcześniej w tym sprawozdaniu.

```
#...
.text
.global _start
_start:
call processInputNumber #wczytaj i przetworz pierwsza liczbe
push %rdx
call factorial
  call processOutputNumber
jmp out
```

```
factorial:
push %rbp
mov %rsp, %rbp  # nowy wskaznik ramki
mov 16(%rbp), %rax  # pobranie parametru z wnetrza stosu
cmp $1, %rax # Sprawdzenie warunku zatrzymania
je factorial_end

dec %rax # zmniejsz licznik poziomu iteracji
push %rax
call factorial # wywolaj funkcje rekurencyjnie

mov 16(%rbp), %rbx #pobierz aktualny poziom rekurencji
mul %rbx #oblicz iloczyn poziomu rekurencji


factorial_end:
mov %rbp, %rsp # przywroc wskaznik stosu
pop %rbp  #
ret  # Return
```

2.3 Wnioski

Poprawne operowanie danymi na stosie jest zadaniem nietrywialnym. Początkowo problem sprawiło prawidłowe wyrównywanie stosu. Dopiero dogłębne przeanalizowanie materiałów dotyczących odpowiedniego tworzenia i wykorzystania funkcji pozwoliło na zrealizowanie ćwiczenia.

(3) Zapoznanie z technikami pozwalającymi na użycie w tym samym projekcie różnych języków programowania

3.1 Treść ćwiczenia

Zakres ćwiczenia:

- Użycie języka C w kodzie Assemblera oraz, użycie języka Assemblera w kodzie C. Tworzenie wstawek asemblerowych.

3.2 Przebieg ćwiczenia

3.2.1 Wykorzystanie funkcji bibliotecznych języka C w asemblerze

Pierwszym programem zaimplementowanym na laboratorium była aplikacja wykorzystująca funkcję z biblioteki standardowej języka C, mianowicie *printf* i *scanf*. Podczas używania funkcji *scanf* należy zadeklarować łańcuch formatujący i przekazać go funkcji poprzez stos. Program wczytuje liczbę podana przez użytkownika, następnie wyświetla napis złożony z danych wpisanych w programie, oraz z liczby wprowadzonej przez użytkownikiem

```
.data
.align 32
format_string_input:
.string "%d"

format_string:
.ascii "Przykladowy string,Czesc  %s!, Jestem %s i mam   %d lata \n\0"

text1:
.ascii "Krzysiu\0"   # pierwszy parameter %s (łańcuch znaków zakończony \0)
text2:
.ascii "Rafal\0"   # drugi parameter %s (łańcuch znaków zakończony \0)
number:
.long 0               # trzeci parameter %d (liczba dziesiętna)

.global main
.text

main:

push $number
pushl $format_string_input       #łańcuch formatujący
call scanf
```

```
                            # parametery przez stos w odwróconej kolejności
push number             # trzeci liczba dziesiętna (%d)
pushl $text2                # drugi łańcuch (%s)
pushl $text1                # pierwszy łańcuch (%s)
pushl $format_string        #łańcuch formatujący
call printf

call exit                   # funkcja zakończenia programu
```

Powyższy program został skompilowany następującym poleceniem:

```
gcc zad1.s -m32
```

3.2.2 Użycie funkcji napisanej w asemblerze w programie w języku C

W asemblerze stworzono funkcję przedstawioną na poniższym listingu. Pobiera ona dwa parametry przekazane poprzez stos, następnie dodaje je i wyświetla.

```
SYSCALL32 = 0x80
.data
format_string:
.ascii "Wynik to:   %d \n\0"
.text
.global output

output:
push %ebp
mov %esp, %ebp
mov 8(%ebp), %eax
mov 12(%ebp), %ebx

add %eax, %ebx

push %ebx
push $format_string
call printf

mov %ebp, %esp # przywroc wskaznik stosu
pop %ebp  #

ret
```

Funkcja została wywołana w kodzie napisanym w języku C. Kod został przedstawiony poniżej.

```
#include <stdio.h>

void output(int a, int b);

int main()
```

```
{
    output(2,3);
    return 0;
}
```

Powyższym program został skompilowany poprzez poniższy Makefile:

```
# reguła linkowania
zad2: output.o zad2.o
    ld output.o zad2.o -o zad2
# reguła kompilacji
output.o: output.s
    as output.s -o output.o
zad2.o: zad2.c
    gcc zad2.c -o zad2.o
clean:
    rm -f *.o
run:
    ./zad2
```

Program skompilowano i uruchomiono. Zaobserwowano prawidłowe działanie. Początkowo wystąpiły trudności w prawidłowym stworzeniem funkcji w asemblerze, tak by mogła odbierać parametry poprzez stos. Było to spowodowane nieprawidłowym wyliczaniem adresu przekazywanych parametrów na stosie. Ponadto należało usunąć etykietę *main* z pliku asemblerowego, prowadziło to do niejednoznaczności podczas kompilacji.

3.2.3 Użycie funkcji napisanej w C w programie asemblera

Program ma działanie analogiczne do poprzedniego. Tym razem asembler jest odpowiedzialny za wczytanie dwóch liczb, przekazanie ich jako parametrów oraz wywołanie funkcji wykonującej dodawanie i wyświetlenie wyniku. W tym przypadku etykieta *main* musi znajdować się z programie asemblerowym.

```
.data
.align 32
format_string_input:
.string "%d %d"
number1:
    .long 0            # jeden parameter %d (liczba dziesiętna)
number2:
    .long 0            # drugi parameter %d (liczba dziesiętna)

.global main
.text
main:

push $number1
push $number2
pushl $format_string_input       #łańcuch formatujący
```

```
call scanf

push number1
push number2
call dodaj

call exit                   # funkcja zakończenia programu
```

Poniżej przedstawiono kod programu w języku C, który jest wywoływany w programie asemblerowym. Pobiera on dwa argumenty, będące liczbami całkowitymi, następnie dodaje je i wypisuje.

```
#include <stdio.h>
void dodaj(int a, int b){
printf("%d", a+b);
}
```

Program został skompilowany poleceniem:

```
gcc zad2.c add.s -m32
```

3.2.4 Wstawka asemblerowa

Zaimplementowano wstawkę dodającą dwie liczby.

```
#include <stdio.h>

int main()
{
int a = 3;
int b = 5;
int result = 0;
__asm__(
"mov %1, %%ecx;\n"
        "mov %2, %%ebx;\n"
"add %%ecx,%%ebx\n"
"mov %%ebx,%0\n"
         : "=b" ( result )        /* output */ parametry wyjsciowe
         : "c" ( a ), "b" (b)     /* input *///przekazywnie parametrow do wstawki
         : "%ecx" //informacja o niszczonych rejestrach
);

printf("%d",result);
return 0;
}
```

Kod został skompilowany poniższym poleceniem:

```
gcc zad3.c
```

3.3 Wnioski

Umiejętność łączenia programów napisanych w C i asemblerze jest praktyczna. Odpowiednie przekazywanie parametrów poprzez stos wymaga dogłębnego zrozumienia prawidłowej obsługi zmiennych odkładanych na stosie.

4.1 Treść ćwiczenia

Zakres ćwiczenia:

- Zapoznanie się z działaniem jednostki zmiennoprzecinkowej, jej kodami błędów oraz możliwymi działaniami które można na niej wykonać..

4.2 Przebieg ćwiczenia

Pierwszym krokiem było stworzenie aplikacji umożliwiającej odczyt odpowiednich flag błędów, oraz wyświetlania komunikatów z tym związanych na ekran. Kolejne polecenia dotyczyły modyfikacji istniejącego już kodu. W wyniku tego rezultatem ćwiczenia jest jedna aplikacja. Kod występujący po etykiecie *get-error* sekwencyjnie sprawdza stan rejestru stanu jednostki zmiennoprzecinkowej. Ostatnie 6 bitów zawiera flagi informujące użytkownika o tym, czy dany błąd wystąpił, czy nie.
Kolejnym poleceniem było przetestowanie działania jednostki podczas operacji prowadzących do sytuacji zmieniających flagi w rejestrze stanu. Próbowano np. podzielić liczbę przez 0, czy obliczyć pierwiastek z liczby ujemnej.
Ostatnim etapem było wykonanie obliczeń na liczbach. Dodano fragment mnożący dwie liczby zmiennoprzecinkowe przez siebie.

```
SYSCALL32 = 0x80
WRITE = 4
STDOUT = 1
EXIT = 1
.data
.align 32
status_word: .int 128
control_word: .int 128
PE:
    .string "Blad precyzj\n"
UE:
    .string "Niedomiar\n"
OE:
    .string "Nadmiar\n"
ZE:
    .string "Zero divide\n"
DE:
    .string "zdenormalizowany operand precyzji\n"
IE:
    .string "niepoprawna instrukcja\n"
```

```
value1:
    .float 2.46
value2:
    .float 5.45
wynik:
    .space 8  #zmienna tymczasowa

.text
.global main

main:
finit #inicjacja kooprocesora

fstcw control_word       #rejestr sterowania
mov control_word, %eax   #
or $4, %eax      #ustawienie blokowanie wyjatku dzielenia przez 0

#dzielenie przez 0
fld1 #zaladownie liczby 1
fldz #zaladownie liczby 0
fdivrp #dzielenie przez 0

#pierwiastek z ujemnej
fld1 #zaladowanie liczby 1
fchs #zmiana znaku
fsqrt #pierwiastkowanie

#mnozenie dwoch liczb
fld value1
fmul value2
#fst  %st(0)

fstsw status_word       #informacje o wystepujacych wyjatkach
mov status_word, %edi   #znajduja sie na najmlodszych 6 bitach slowa kontrolnego


get_error: #rozpoznaje bledy z rejestru eax

test_IE:
test $1, %edi
jz test_DE
pushl $IE
call printf

test_DE:
test $2, %edi
jz test_ZE
pushl $DE
```

```
call printf

test_ZE:
test $4, %edi
jz test_OE
pushl $ZE
call printf

test_OE:
test $8, %edi
jz test_UE
pushl $OE
call printf

test_UE:
test $16, %edi
jz test_PE
pushl $UE
call printf

test_PE:
test $32, %edi #testowanie bitu na 5 pozycji
jz exit
pushl $PE
call printf

exit:
movl  $EXIT,%eax
int $SYSCALL32
```

Kod skompilowano następującym poleceniem:

```
gcc zad.c add.s -m32
```

4.3 Wnioski

Płynne posługiwanie się jednostką zmiennoprzecinkową wymaga dobrej znajomości listy rozkazów oraz architektury. Pomocne w przygotowaniu do laboratorium jest zapoznanie się z budową rejestru sterującego i kontrolnego.