Aleksandra Ojdana, 198842
PT-N-17

Wrocław, 08.04.2019

Prowadzący: dr Piotr Patronik

Laboratorium Organizacja i Architektura Komputerów
Podstawowe operacje arytmetyczne, I/O standardowego wejścia/wyjścia, zawartość stosu

1. Treść ćwiczenia

1.1 Zakres i program ćwiczenia

Celem ćwiczenia była realizacja następujących programów w języku asembler:

- wykonującego podstawowe operacji arytmetycznych (dodawanie, odejmowanie, mnożenie, dzielenie) dla liczb o precyzji powyżej 64 bitów,
- drukującego na standardowe wyjście parametry swojego wywołania, w tym ścieżkę do programu oraz zawartość środowiska,
- pobierający cyfry w systemie z bazą 16 o precyzji powyżej 64 bitów ze standardowego wyjścia oraz wyświetlający pobraną liczbę na standardowe wyjście,
- pobierający cyfry w systemie z bazą 10 o precyzji powyżej 64 bitów ze standardowego wyjścia oraz wyświetlający pobraną liczbę na standardowe wyjście,
- program wczytujący liczby z wykorzystaniem opcji: int $0x80, printf/scanf, podanych jako parametry programu.

W trakcie ćwiczeń obowiązywała znajomość poniżej wymienionych komend debuggera, które poza znajomością, należało również umieć zastosować.
Wymagane komendy debuggera:

- dissassemble (w tym disassemble /r, disassemble _start, disassemble _start +10, itp.)
- breakpoint, delete, watch, condition
- step, stepi, next, nexti, cont, finish
- info registers, info all-registers, print, display
- x (w tym x/Nbx adres, x /Nxf, x/Ni _start, itp.)
- dump (wraz z następującym wypisaniem zawartości przy użyciu programu hexdump -C)

Dodatkowo na ćwiczeniach wymagana była umiejętność omówienia zagadnień dotyczących:

- przekazywanie parametrów do funkcji bibliotecznych
- zaprezentowania ramki stosu dla wywołania funkcji bibliotecznej, w tym przekazywania parametrów, na poziomie gdb, z użyciem komendy x
- zawartość stosu w momencie uruchomienia funkcji _start

1.2 Zadania zrealizowane podczas ćwiczeń

- program wyświetlający w konsoli parametry swojego wywołania oraz zawartość środowiska
- program dodający liczby o precyzji powyżej 64 bitów
- program odejmujący liczby o precyzji powyżej 64 bitów
- poznanie oraz użycie obowiązujących komend debuggera
- zapoznanie się z dodatkowymi zagadnieniami: przekazywanie parametrów do funkcji bibliotecznych i prezentowanie ramki stosu na poziomie gdb

1.3 Zadania zrealizowane poza ćwiczeniami

- program pobierający cyfry w bazie 16, ze standardowego wejścia i wypisujący je na standardowe wyjście
- program pobierający cyfry w bazie 10, ze standardowego wejścia i wypisujący je na standardowe wyjście
- program wczytujący liczby wykorzystujący printf/scanf
- program wczytujący liczby podane jako parametry programu

2 Przebieg ćwiczenia

2.1 Zagadnienia teoretyczne

Przed przystąpieniem do realizacji programów zapoznano się z teoretycznymi zagadnieniami opisanymi w punkcie 1.1.

1. Zawartość stosu w momencie uruchomienia funkcji _start.

W momencie uruchomienia funkcji _start na stosie znajdują się kolejno: ścieżka wykonywanego programu, parametry przekazane do programu, zmienne środowiskowe. Zawartość stosu sprawdzono empirycznie.

2. Wczytywanie cyfr ze standardowego wejścia i wyświetlanie ich na standardowe wyjście.

Liczby wczytane ze standardowego wejścia do bufora, wczytywane są jako znaki ASCII zapisane w systemie szesnastkowym. Przykładowo podając cyfrę ‘8’, której kod ASCII jest równy 56 w systemie dziesiętnym, do bufora zostanie zapisana jego wartość w systemie szesnastkowym czyli 0x38. Zapis jednej cyfry odbywa się na jednym bajcie, zapisując do bufora liczbę np. 5678, pod kontrolą gdb można sprawdzić zawartość tego bufora np. poleceniem x/8bx adres_bufora → adres_bufora: 0x35 0x36 0x37 0x38 0x00 0x00 0x00 0x00. Chcą wykonać operację arytmetyczne na wczytywanych liczbach z konsoli, należy je najpierw przekształcić na jej odpowiednik w systemie szesnastkowym, w tym przypadku byłaby to wartość 0x162E. Dopiero na takiej liczbie można wykonać operację arytmetyczną. Następnie wynik takiej operacji przed wyświetleniem na ekranie należy przekształcić na reprezentację w kodzie ASCII.

3. Przekazywanie parametrów do funkcji bibliotecznych

Przekazywanie parametrów do funkcji bibliotecznych odbywa się w odwróconej kolejności tzn. ostatni parametr funkcji przekazuję się jako pierwszy poprzez wrzucenie jego wartości na stos poleceniem pop, pierwszy parametr funkcji przekazuje się jako ostatni. Przekazywanie parametrów do funkcji odbywa się poprzez umieszczanie ich na stosie, bezpośrednio przed wywołaniem funkcji.

4. Prezentacja ramki stosu dla wywołania funkcji bibliotecznej

5. Obowiązujące komendy debuggera

Opisy komend oraz sposób ich wykorzystania sprawdzano poleceniem help w debugerze.

disasseble - wykonuje zrzut pamięci wczytanego programu w danym miejscu, po komendzie podaje się nazwę funkcji lub adres w pamięci
flaga /r do disassemble - dodanie surowych instrukcji w systemie szesnastkowym

breakpoint - ustawienie w programie breakpointa, po poleceniu należy podać numer linii lub nazwę funkcji
delete - usuwa wcześniej ustawione breakpointy
watch - ustawienie breakpointa na zmiennej, za każdym razem, gdy wartość zmiennej zostanie zmieniona program się zatrzyma i wyświetli wartość zmiennej
condition – ustawia dla wskazanego breakpointa warunek, który musi zostać spełniony, gdy program dojdzie do wskazanego breakpointa
step - po zatrzymaniu programu, funkcja step uruchamia następny krok programu, jeżeli jest to funkcja to nie wchodzi do środka funkcji
stepi - tak samo jak step, z tym, że w przypadku trafienie na funkcję, wchodzi do niej
next – przejście programu do następnej linii, nie wchodzi do funkcji
nexti – przejście programu do następnej linii, wchodzi do funkcji
continue – kontynuuje debugowanie do następnego breakpointa, przerwania lub końca
finish – kończy debugowanie funkcji, w której się znajduje
info registers – wyświetla zawartość rejestrów
info all-registers – wyświetla zawartość rejestrów oraz flag
print – wyświetla zawartość pod wskazanym adresem
display – wyświetla zawartość pod wskazanym adresem, za każdym razem jak program się zatrzyma
x/Nbx adres – wyświetla zawartość danego adresu w formacie szesnastkowym, N odpowiada liczbie wyświetlonych bajtów
x/Nwf - wyświetla zawartość danego adresu w formacie float, N odpowiada liczbie słów maszynowych
x/Ni - wyświetla zawartość danego adresu w formie instrukcji
dump – wykonuje zrzut pamięci i zapisuje jej zawartość od wskazanego pliku

2.2 Uruchomienie programu

Procedura uruchamiania wszystkich programów z wyjątkiem jednego wymagającego użycia funkcji bibliotecznych odbyła się na takiej samej zasadzie jak podczas wykonywania ćwiczeń na poprzednich zajęciach, z wykorzystaniem pliku Makefile. Ta część ćwiczeń została opisana w sprawozdaniu do laboratorium „Podstawy uruchomienia programów asemblerowych na platformie Linux/x86”.
Stworzono dwa programy wykorzystujące funkcję umieszczone w innym pliku niż program. Pierwszy z nich napisano w języku asembler NASM i wykorzystano standardowe biblioteki, drugi w notacji AT&T z wykorzystaniem funkcji z własnych plików. Drugi program z notacji AT&T uruchomiono analogicznymi poleceń z makefile:

as --32 -g pr_sc.s -o pr_sc.o
ld -melf_i386 --o pr_sc pr_sc.o -lc

Do uruchomienia pierwszego napisanego w NASM wykorzystano polecenia [1]:

nasm -f elf -o pr_sc.o pr_sc.asm
gcc -s -o pr_sc pr_sc.o

Wcześniej wymagane było pobranie odpowiedniego kompilatora NASM.

2.3 Konstrukcja pliku źródłowego

Konstrukcja plików źródłowych wyglądała podobnie jak w plikach źródłowych z poprzednich ćwiczeń. Wyjątek stanowiły pliki źródłowe do zadania z funkcjami bibliotecznymi. Różni się on od pozostałych tym, że na początku wymagana jest dyrektywa .**include**[2], w której umieszcza się zewnętrzne pliki źródłowe lub dodaniem **global printf/scanf** w NASM [1].

Dodatkowe rzeczy jakie pojawiły się w plikach źródłowych w tym ćwiczeniu, a nie występowały w ćwiczeniu poprzednim to:

1. Dyrektywy [2]:
    - .include: wskazanie sekcji zawierającej odniesienia do plików zewnętrznych
    - .bss: wskazanie sekcji zawierającej deklarację buforów
    - .space: deklarowanie miejsca w pamięci na bufor, należy podać długość w bajtach, opcjonalnie wartości jakimi bufor zostanie domyślnie wypełniony
    - .long: deklarowanie miejsca w pamięci na wartość typu long, domyślnie 4 bajty
    - .type: oznaczenie typu wartości umieszczonej pod wskazaną etykietą
2. Funkcje [1]:
    - pop – umieszczenie wartości na stosie
    - push – pobranie wartości ze stosu
    - clc – zerowanie flagi przeniesienia
    - addcl – sumowanie wartości A i B i flagi CF
    - pushf – umieszczenie na stosie flag
    - popf – pobranie ze stosu wartości do flag
    - xor – wykonanie operacji xor na A i B
    - call – wywołanie funkcji
3. Inne [3]
    - @function – określenie typu jako funkcja

3. Wnioski

Zaimplementowano poprawnie programy: drukujący na standardowe wyjście parametry swojego wywołania i zawartość środowiska, program dodający i program odejmujący liczby o dowolnej precyzji, program stanowiący wyjście\wejście do programów dodającego i odejmującego, a także program wczytujący liczby z wykorzystaniem funkcji printf i scanf i program wczytujący liczby podane jako parametry programu. Nie zrealizowano programu mnożącego/dzielącego liczby o dowolnej precyzji.
Podczas realizacji zadań problemem okazało się wykorzystanie funkcji ze standardowych biblioteki Linuxa („linux.s”). Pomimo zastosowania się do zaleceń zawartych w książce „Programming from the Ground Up” w rozdziale 8 „Sharing Functions with Code Libraries” nie udało się napisać programu w notacji AT&T wykorzystującego standardowe funkcję printf i scanf, dlatego zdecydowano się na rozwiązanie tego problemu poprzez napisanie programu w notacji NASM oraz napisanie własnej implementacji funkcji printf i scanf.

4. Literatura

1. „Asembler x86/Łączenie z językami wysokiego poziomu/Funkcje zewnętrzne”, https://pl.wikibooks.org, dnia 08.04.2019
2. https://sourceware.org/binutils/docs-2.32/ld/index.html dn. 08.04.2019
3. J. Bartlett “Programming from the Ground Up” GNU Free Documentation License

1. Program drukujący na standardowe wyjście parametry swojego wywołania oraz zawartość środowiska.

```
EXIT =1
WRITE =4
STDOUT =1
RETURN =0

.align 32
.section .data
new_line: .ascii "\n"
new_line_len = . - new_line

.global _start
_start:
pop %ecx
mov $8, %edx
pop %ecx
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80

mov $new_line_len, %edx
mov $new_line, %ecx
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80

pop %ecx
mov $0, %edi

loop_start:
cmpl $62, %edi
je loop_end
inc %edi
mov $32, %edx
pop %ecx
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80
jmp loop_start

loop_end:
mov $new_line_len, %edx
mov $new_line, %ecx
mov $STDOUT, %ebx
mov $WRITE, %eax
int $0x80

mov $RETURN, %ebx
mov $EXIT, %eax
int $0x80
```

2. Program dodający liczby o dowolnej precyzji

```
SYSEXIT = 1
EXIT_SUCCESS = 0
.data
liczba1: .long 0xF1111111, 0xE2222222, 0x33333333, 0x04444444
liczba2: .long 0x05555555, 0x44444444, 0x33333333, 0x0F222222

#Liczba iteracji pętli
iterator = ((. - liczba1)/4)-1
.global _start
_start:
#Zerowanie flagi przeniesienia
clc
#Ustawienie iteratora dla pętli
movl $iterator, %ecx

loop:
#Wpisanie liczb do rejestrów
movl liczba1(,%ecx,4), %eax
movl liczba2(,%ecx,4), %ebx

#Dodawanie adcl - sumuje A i B i flage CF
adcl %ebx, %eax

#Wysłanie wyniku do stosu
pushl %eax

pushf

#Zmniejszenie licznika pętli
subl $1, %ecx
cmp $0, %ecx
jl last
popf
jmp loop

#Dopisanie 1 przy ostatnim przeniesieniu
last:
popf
jnc end
xor %edx, %edx
movl $1, %edx
pushl %edx

end:
mov $SYSEXIT, %eax
mov $EXIT_SUCCESS, %ebx
int $0x80
```

3. Program odejmujący liczby o dowolnej precyzji

```
SYSEXIT = 1
EXIT_SUCCESS = 0
.data
liczba1: .long 0xF1111111, 0xE2222222, 0x33333333, 0x04444444
liczba2: .long 0x15555555, 0x44444444, 0x33333333, 0x0F222222

iterator = ((. - liczba1)/4)-1

.global _start
_start:
#Wyczyszczenie flagi przeniesienia
clc

#Ustawienie iteratora dla pętli
movl $iterator, %ecx

pushf

loop:
popf

#Wpisanie liczb do rejestrów
movl liczba1(,%ecx,4), %eax
movl liczba2(,%ecx,4), %ebx

#Odejmowanie
sbbl %ebx, %eax

#Wysłanie wyniku do stosu
pushl %eax
pushf

#Zmniejszenie licznika pętli
subl $1, %ecx
cmp $0, %ecx
jl last
jmp loop

#Odjęcie 1 przy ostatniej pożyczce
last:
popf
jnc end
xor %eax, %eax
subl $1,%eax
pushl %eax

end:
mov $SYSEXIT, %eax
mov $EXIT_SUCCESS, %ebx
int $0x80
```

4. Wejście/wyjście dla programów nr 2 i 3 liczb w bazie 10

```
EXIT =1
WRITE =4
READ =3
STDIN =0
STDOUT =1
RETURN =0
BUFF_SIZE =100

.bss
numberOnein: .space 100
numberOneHex: .space 100
numberTwoHex: .space 100
numberTwoin: .space 100


.data
promptFirst: .ascii "Podaj pierwsza liczbe\n"
promptFirst_len = . - promptFirst
promptSecond: .ascii "Podaj druga liczbe\n"
promptSecond_len = . - promptSecond
wrong_format_text: .ascii "Niepoprawny format danych\n"
wrong_format_text_len = . - wrong_format_text
sumup: .ascii "Podane liczby to: "
sumup_len = . - sumup

.global _start
_start:

#"Podaj pierwsza liczbe"
mov $promptFirst_len, %edx
mov $promptFirst, %ecx
mov $WRITE, %eax
mov $STDOUT, %ebx
int $0x80

#"Wprowadzanie pierwszej liczby"
mov $numberOnein, %ecx
mov $BUFF_SIZE, %edx
mov $READ, %eax
mov $STDIN, %ebx
int $0x80

mov $0, %eax
mov $0, %ebx

#Sprawdzenie formatu pierwszej liczby
mov $0, %edi
check_no_one_start:
cmp $BUFF_SIZE, %edi
je end_checking_no_one
```

```
mov numberOnein(, %edi,1), %bl
inc %edi
#znak nowej linii wczytany do bufora = 0x0a czyli 10 w dec
cmp $10, %bl
je end_checking_no_one

cmp $48, %bl
jl wrong_format
cmp $57, %bl
jg wrong_format
jmp check_no_one_start

end_checking_no_one:
#"Podaje druga liczbe"
mov $promptSecond_len, %edx
mov $promptSecond, %ecx
mov $WRITE, %eax
mov $STDOUT, %ebx
int $0x80

#"Wprowadzanie drugiej liczby"
mov $numberTwoin, %ecx
mov $BUFF_SIZE, %edx
mov $READ, %eax
mov $STDIN, %ebx
int $0x80

#Sprawdzenie formatu drugiej liczby
mov $0, %edi

check_no_two_start:
cmp $BUFF_SIZE, %edi
je end_checking_no_two

mov numberTwoin(, %edi,1), %bh
inc %edi
#znak nowej linii wczytany do bufora = 0x0a czyli 10 w dec
cmp $10, %bh
je end_checking_no_two

cmp $48, %bh
jl wrong_format
cmp $57, %bh
jg wrong_format

jmp check_no_two_start

end_checking_no_two:

#"Podane liczby to: "
mov $sumup_len, %edx
mov $sumup, %ecx
```

```
mov $WRITE, %eax
mov $STDOUT, %ebx
int $0x80

#"wypisanie pierwszej liczby "
mov $BUFF_SIZE, %edx
mov $numberOnein, %ecx
mov $WRITE, %eax
mov $STDOUT, %ebx
int $0x80

#"wypisanie drugiej liczby "
mov $BUFF_SIZE, %edx
mov $numberTwoin, %ecx
mov $WRITE, %eax
mov $STDOUT, %ebx
int $0x80

jmp program_end
# Wrong format
wrong_format:
mov $0, %ecx
mov $0, %edx
mov $wrong_format_text_len, %edx
mov $wrong_format_text, %ecx
mov $WRITE, %eax
mov $STDOUT, %ebx
int $0x80

program_end:
mov $RETURN, %ebx
mov $EXIT, %eax
int $0x80
```

5. Wejście/wyjście dla programów nr 2 i 3 liczb w bazie 16

Tak jak program nr 4, różnica w sprawdzeniu podanych liczb:

```
check_no_two_start:
cmp $BUFF_SIZE, %edi
je end_checking_no_two

mov numberTwoin(, %edi,1), %bl
inc %edi
#znak nowej linii wczytany do bufora = 0x0a czyli 10 w dec
cmp $10, %bl
je end_checking_no_two

cmp $48, %bl
jl wrong_format
cmp $70, %bl
jg wrong_format
```

```
cmp $57, %bl
jg check_if_letter_two
jmp check_no_two_start

check_if_letter_two:
cmp $65, %bl
jl wrong_format
jmp check_no_two_start

end_checking_no_two:
```

6. Program wykorzystujący funkcję printf i scanf – NASM [źródło: „Programming Intel i386 Assembly with NASM”, Y. Hardy]

```
section .data
    message1: db "Podaj pierwsza liczbe: ", 0
    message2: db "Podaj druga liczbe: ", 0
    formatin: db "%d", 0
    formatout: db "%d", 10, 0
    integer1: times 4 db 0
    integer2: times 4 db 0
section .text
   global main
   extern scanf
   extern printf

main:

   push ebx
   push ecx
   push message1
   call printf

   add esp, 4 ; sciaganie argumentow funkcji printf ze stosu
   push integer1
   push formatin
   call scanf

   add esp, 8 ; sciaganie argumentow funkcji scanf ze stosu
   push message2
   call printf

   add esp, 4     ; sciaganie argumentow funkcji printf ze stosu
   push integer2
   push formatin
   call scanf

   add esp, 8 ; sciaganie argumentow funkcji scanf ze stosu

   mov ebx, dword [integer1]
   mov ecx, dword [integer2]
```

```
  push ebx
  mov ebx, ecx
  push formatout
  call printf           ; wyswietlanie pierwszej podanej liczby
  add esp, 8
  push ebx
  push formatout
  call printf            ; wyswietlanie drugiej podanej liczby
  add esp, 8
  pop ecx
  pop ebx ; odzyskiwanie wartości rejestrów
  mov eax, 0 ; poprawne wyjscie z programu
  ret
```

7. Program wykorzystujący funkcję printf i scanf – własna implementacja

```
.include "stdio.s"
.bss
numberOne: .space 4
numberTwo: .space 4

.data
promptOne: .ascii "Podaj pierwsza liczbe: \n\0"
promptTwo: .ascii "Podaj druga liczbe: \n\0"
promptTwo_len = . - promptTwo

.global _start
_start:

push $promptOne
push $promptOne_len
call printf
add  $8, %esp

push $numberOne
push $4
call scanf
add  $8, %esp

push $promptTwo
push $promptTwo_len
call printf
add  $8, %esp

push $numberTwo
push $4
call scanf
add  $8, %esp

mov $0, %ebx
mov $1, %eax
int $0x80
```

8. Zawartość pliku stdio.s

```
.type printf, @function
printf:
push %ebp
mov %esp, %ebp

#"Wypisanie tekstu"
mov 8(%ebp), %edx
mov 12(%ebp), %ecx
mov $4, %eax
mov $1, %ebx
int $0x80

mov %ebp, %esp
pop %ebp
ret

.type scanf, @function
scanf:
push %ebp
mov %esp, %ebp

#"Pobranie tekstu"
mov 8(%ebp), %edx
mov 12(%ebp), %ecx
mov $3, %eax
mov $0, %ebx
int $0x80

mov %ebp, %esp
pop %ebp
ret
```