Wrocław, dn. 15 maja 2016r

Jakub Pomykała, 209897
PN-P-8

prowadzący: prof. Janusz Biernat

Laboratorium Architektury Komputerów
(0) Podstawy uruchamiania programów asemblerowych
na platformie Linux/x86

0 Treść ćwiczenia

Zakres ćwiczenia:

- Zapoznanie się z podstawami pisania i uruchamiania programów w języku assembler.
- Nauka obsługi środowiska programistycznego `vim`.
- Zapoznanie się z narzędziem `gdb`.

Zrealizowane zadania:

- Uruchomienie pierwszego programu
- Wczytanie znaków z klawiatury do bufora
- Operacje na wczytanych znakach
- Wyświetlenie znaków z bufora
- Zapoznanie się z debuggerem `gdb`

0.1 Przebieg ćwiczenia

Kod programu został skopiowany z [1]. Jest to podstawowy szkielet programu, który był wykorzystywany w następnych programach. Ponizszy kod wyświetla na ekranie komputera napis "Hello World".

```
SYSCALL = 0x80 # nr przerwania - wywołanie systemowe
EXIT = 1 # nr funkcji - wyjście z programu
WRITE = 4 # nr funkcji - wypisywanie danych
READ = 3 # nr wyjscia - stadnardowego
STDOUT = 1 # nr wyjscia - ekran
STDIN = 0 # nr wyjścia  - klawiatura

.data
informacja: .ascii "Hello world\n" # ciag znakow do wyswietlenia
rozmiar_informacja = . - informacja # obliczenie dlugosci ciagu znakow

.text
.global _start # etykieta od ktorej program zaczyna wykonanie
```

```
_start:
#wypisywanie tekstu
movl $WRITE,  %eax # funkcja - wypisz
movl $STDOUT,  %ebx # numer wyjscia - ekran
movl $informacja,  %ecx # adres ciagu znakow do wypisania
movl $rozmiar_informacja,  %edx # ilosc bajtow do wypisania
int  $SYSCALL # wywolanie przerwania systemowego, w celu wykonania funkcji wypisz

#poprawne wyjscie z programu
movl $EXIT, %eax # funkcja - koniec programu
int  $SYSCALL # zakonczenie programu
```

W celu uruchomienia powyższego programu należało go skompilować, a następnie skonsolidować przy pomocy poniższych instrukcji.

```
#kompilacja
as lab.s -o lab.o

#konsolidacja
ld lab.o -o lab

#uruchomienie
./lab
```

Powyższe instrukcje zostały wykorzystane do stworzenia skryptu o nazwie `makefile.sh` w celu przyspieszenia pracy.

0.2 Wczytywanie znaków z klawiatury do bufora

Program miał za zadanie zapisać znaki wprowadzone przez użytkownika do bufora o nazwie `bufor`.

```
SYSCALL = 0x80
EXIT = 1
WRITE = 4
READ = 3
STDOUT = 1
STDIN = 0

buf_size = 31
text_size: .long 0
bufor: .space buf_size

.text
.global _start

_start:
#wczytywanie znaków z klawiatury do bufora
movl $READ, %eax
movl $STDIN, %ebx
movl $bufor, %ecx
movl $buf_size, %edx
int  $SYSCALL
```

```
#rzeczywista liczba wpisanych znakow w text_size
movl %eax, text_size
```

0.3 Wyświetlanie znaków z bufora

Program ma za zadanie wyświetlić na ekranie komputera znaki z bufora.

```
.data
buf_size = 31
text_size: .long 0
bufor: .space buf_size

.text
.global _start

_start:
#wyswietlenie znaków z bufora
WYSWIETL:
movl $WRITE, %eax # funkcja zapisu na ekran
movl $STDOUT, %ebx # wyswietlenie na ekranie
movl $bufor, %ecx # tekst to wyświetlenia
movl $buf_size, %edx # rozmiar tekstu
int  $SYSCALL
```

0.4 Debugowanie programów przy pomocy `gdb`

`gdb` to narzędzie, które pozwala na ustawianie pułapek w kodzie programu w celu kontrolowania wartości znajdujących się rejestrach. Aby skorzystać z możliwości programu `gdb` należy w kodzie ustawić pułapki poprzez dodanie etykiet. Przyjęta została konwencja, że etykiety dla `gdb` będą poprzedzone literą `t` od słowa trap oraz kolejną cyfrą, np.: `t0`. Pułapki programu są ustawiane w miejscach, w których wymagane jest skontrolowanie wartości, znajdujących się aktualnie w rejestrach procesora. Po ustawieniu pułapek w kodzie programu należy skonsolidować oraz zlinkować kod do postaci wykonywalnej przy pomocy komend:

```
as -o lab.o lab.s
ld -o lab lab.o
```

Po poprawnym zlinkowaniu programu należy uruchomić go pod kontrolą gdb

```
gdb ./lab
```

Najważniejszymi i najbardziej przydatnymi komendami w programie `gdb` są:

- `info r` – wyświetlenie informacji o zawartości rejestrów w danym momencie wykonywania programu,
- `break` – ustawienie pułapki,
- `run` – uruchomienie programu,
- `next` – wykonywanie programu do kolejnej pułapki lub końca programu.

0.5 Wnioski

Pierwsze zajęcia laboratoryjne pozwoliły na zapoznanie się ze składnią programów w języku assembler. Nowo poznane operacje wejścia/wyjścia są podstawową metodą komunikacji użytkownik-komputer. Dzięki narzędziu `gdb` istnieje prosty sposób na analizę krokową wykonywanego kodu programu. W celu przyspieszenia tworzenia prograów poszczególne elementy kodu źródłowego będą używane na kolejnych laboratoriach.

1 Treść ćwiczenia

Zakres ćwiczenia:

- Przetwarzanie tekstu
- Konwersje liczbowe
- Korzystanie z podstawowych instrukcji logicznych języka assembler

Zrealizowane zadania:

- Program zamieniający wielkość liter
- Program szyfrujący/deszyfrujący - szyfr Cezara
- Konwersja liczb w systemie szesnastkowym na wartość w rejestrze
- Konwersja liczb w systemie dziesiętnym na wartość w rejestrze przy pomocy schematu Hornera
- Wyświetlanie liczby w rejestrze w postaci dziesiętnej, szesnastkowej oraz dwójkowej przy pomocy schematu Hornera

1.1 Przebieg ćwiczenia

Program zamieniający wielkość liter

Program ma za zadnie zamieniać małe litery na wielkie oraz wielkie na małe. Zostało to zrealizowane przy pomocy użyciu maski `0x20` oraz operacji logicznej `xorb`, która realizuje funkcję alternatywy wykluczającej.

```
.data
buf_size = 31
text_size: .long 0
bufor: .space buf_size
.text
.global _start

_start:
#wczytwanie z klawiatury do bufora
movl $READ, %eax
movl $STDIN, %ebx
movl $bufor, %ecx
movl $buf_size, %edx
int  $SYSCALL

movl %eax, text_size #liczba wpisanych znakow w text_size
movl $-1, %esi #wstawienie 1 na 32bitach

#pętla która iteruje po kazdym znaku w buforze
```

```
_petla:
incl %esi #zwiekszenie licznika
movb bufor(,%esi, 1), %al  #w rejestrze al, znajduje sie znak z bufora

#sprawdzenie to już koniec znaków w buforze
#jesli rejestr al zawiera znak konca linii '\n' to koniec
cmp $'\n', %al
je WYSWIETL

xorb $0x20, %al #maska dla zamiany litery
movb %al, bufor(,%esi, 1) #zastapienie aktualnego znaku w buforze
jmp _petla # powrót do odczytywania kolejnego znaku

#wyswietlenie z bufora
WYSWIETL:
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $bufor, %ecx
movl $buf_size, %edx
int  $SYSCALL

#poprawne wyjscie z programu
movl $EXIT, %eax
int  $SYSCALL
```

Program został w całości napisany na zajęciach laboratoryjnych. Nie zostały napotkane żadne problemy przy jego realizacji.

Program szyfrujący/deszyfrujący- szyfr Cezara

Poniższy program jest programem szyfrującym oraz deszyfrującym. Zastosowano metodę szyfrowania Jest to zmodyfikowana wersja programu z poprzednich laboratoriów, a główna zmiana polegała na sposobie operacji na pojedynczym znaku z bufora. Oprócz tego została dodana opcja wprowadzania klucza szyfracji/deszyfracji. Mała litera alfabetu służy do szyfracji, wielka do deszyfracji.

```
LICZBA_LITER_ALFABETU = 'Z'-'A'-1
ZERO = 0
DUZA_LITERA = 0x20

.data
#informacja z prosba o wprowadzenie tekstu do zaszyfrowania
prosba: .ascii "Prosze podac tekst: "
rozmiar_prosba = . - prosba

#informacja z prosba o wprowadzenie klucza
prosba_o_klucz: .ascii "Prosze podac klucz: "
rozmiar_prosba_o_klucz = . - prosba_o_klucz

#bufor do przechowywania klucza
klucz_buf_size = 7
klucz_size: .long 0
klucz_bufor: .space klucz_buf_size
```

```
#bufor do przechowywania tekstu do szyfrowania
buf_size = 31
text_size: .long 0
bufor: .space buf_size
.text
.global _start

_start:
#prosba o wpisanie tekstu do zakodowania
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $prosba, %ecx
movl $rozmiar_prosba, %edx
int  $SYSCALL

#wczytywanie z klawiatury do bufora ciagu znakow do zakodowania
movl $READ, %eax
movl $STDIN, %ebx
movl $bufor, %ecx
movl $buf_size, %edx
int  $SYSCALL

movl %eax, text_size #liczba wpisanych znakow w text_size

#prosba o wpisanie klucza
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $prosba_o_klucz, %ecx
movl $rozmiar_prosba_o_klucz, %edx
int  $SYSCALL

#wprowadzenie klucza
movl $READ, %eax
movl $STDIN, %ebx
movl $klucz_bufor, %ecx
movl $klucz_buf_size, %edx
int  $SYSCALL

movl %eax, klucz_size #liczba wpisanych znakow w klucz_size
movl $ZERO, %esi #wyzerowanie licznika, uzywanego do iterowania po buforze

#odczytanie klucza
movb klucz_bufor(,%esi, 1), %bl #w rejestrze bl znajduje się klucz

cmp $'Z', %bl #jesli mala litera to szyfrowanie
ja SZYFROWANIE

DESZYFROWANIE:
subb $'A', %bl #zamiana kodu ASCII na wartosc liczbowa
negb %bl
jmp CEZAR
```

```
SZYFROWANIE:
subb $'a', %bl #zamiana kodu ASCII na wartosc liczbowa

CEZAR:
#przygotowanie do iteracji po znakach do zaszyfrowania
movl $-1, %esi

#petla, ktora iteruje znak po znaku w buforze
_petla:
incl %esi #zwiekszenie licznika
movb bufor(,%esi, 1), %al #w rejestrze al znak z bufora

#sprawdzenie czy jest to litera do zamiany
cmp $'a', %al
jb WYSWIETL
cmp $'z', %al
ja WYSWIETL

#szyfrowanie znaku przy pomocy wprowadzonego klucza
add %bl, %al #dodanie klucza

#sprawdzenie czy nowy znak dalej jest litera w znaczeniu kodu ASCII
cmp $'z', %al
jb WPISZ_DO_BUFORA
#jak znak dalej jest w tablicy ASCII to wpisujemy go od bufora

#naprawia znak ktory jest poza tablica ASCII, po przez odjęcie liczby liter
#alfabetu (działanie zbliżone do funkcji modulo)
NAPRAW_ZNAK:
sub $LICZBA_LITER_ALFABETU, %al

WPISZ_DO_BUFORA:
movb %al, bufor(,%esi, 1) #zastapienie aktualnego znaku w buforze

#sprawdzenie czy koniec bufora
cmp text_size, %esi #czy koniec bufora?
jb _petla #wroc do petli jesli to nie koniec bufora

#wyswietlenie z bufora
WYSWIETL:
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $bufor, %ecx
movl $buf_size, %edx
int  $SYSCALL
```

Program został w całości zrealizowany na zajęciach. Jedynym problem, który się pojawił to informacja o naruszeniu ochrony pamięci w momencie wyścia znaku po za tablicę kodów ASCII. Aby rozwiązać ten prolem należało odjąć liczbę wszystkich znaków w alfabecie.

```
#błędna linijka
sub LICZBA_LITER_ALFABETU, %al
```

```
#poprawiona wersja
sub $LICZBA_LITER_ALFABETU, %al
```

Konwersja liczb w systemie szesnastkowym na wartość w rejestrze

Program konwertuje liczbę podaną w systemie szesnastkowym na wartość liczby. Nie został tutaj użyty schemat Hornera, lecz zastosowano przesunięcie bitowe, możliwe jedynie przy tej konwersji.

```
ODCZYT_WARTOSCI_HEX = 55
ZERO = 0
DUZA_LITERA = 0x20
BREAK_LINE = 2

.data
#bufor do przechowywania wprowadzonych znaków
buf_size = 31
text_size: .long 0
bufor: .space buf_size

.text
.global _start

_start:
#wczytwanie z klawiatury do bufora
movl $READ, %eax
movl $STDIN, %ebx
movl $bufor, %ecx
movl $buf_size, %edx
int  $SYSCALL

movl %eax, text_size #liczba wpisanych znakow w text_size
subl $BREAK_LINE, text_size #ignorowanie ostatnich 2 znaków konca linii '\n'

# START - ZADANIE
movl $ZERO, %eax #miejsce gdzie bedzie przechowywana wartosc
movl $-1, %esi

#petla ktora iteruje znak po znaku
CZYTAJ_ZNAK:
#przesuwa o 4 bity w lewo, na początku nie ma znaczenia, bo sa same zera
shll $4, %eax
incl %esi
movl $ZERO, %ebx #wyzerowanie smieci, dla pewnosci
movb bufor(%esi,1), %bl

#jesli jest to liczba od 0 do 9
cmp $'9', %bl
jbe DEC_DO_BIN

#jesli jest to liczba od A do F
HEX_DO_BIN:
subb $ODCZYT_WARTOSCI_HEX, %bl #odczytanie wartosci z kodu ASCII
```

```
jmp DALEJ

DEC_DO_BIN:
subb $'0', %bl #odczytanie wartosci z kodu ASCII

#wartosc zostaje dodana do rejestru %eax, ktory przechowuje wynik
DALEJ:
addl %ebx, %eax #dodaje

#sprawdzenie czy koniec bufora
cmp text_size, %esi #czy koniec bufora?
jb CZYTAJ_ZNAK #wroc do petli jesli to nie koniec bufora

#pułapka dla GDB, w tym miejscu w %rax znajduje się wartość wpisanej liczby
_t9:

#poprawne wyjscie z programu
movl $EXIT, %eax
int  $SYSCALL
```

Konwersja liczb w systemie dziesiętnym na wartość w rejestrze przy pomocy schematu Hornera

Następny program zamienia znaki dziesiętne (i nie tylko) na wartość w rejestrze przy użyciu schematu Hornera.

```
ZERO = 0
BUFOR_SIZE = 10
BAZA_SYSTEMU = 10 # system dziesietny
.data
buf_size = 31
text_size: .long 0
bufor: .space buf_size
.text
.global _start
_start:

#wczytanie liczby do konwersji
movl $READ, %eax
movl $STDIN, %ebx
movl $bufor, %ecx
movl $buf_size, %edx
int $SYSCALL

mov $ZERO, %edx # rejestr przechowywujący wynik
mov $ZERO, %esi # zerowanie licznika który który iteruje po buforze
mov $ZERO, %ecx # rejestr pomocniczy
mov $ZERO, %ebx # rejestr przechowywujący znak z bufora
mov $ZERO, %eax # rejestr wykorzystywany przy mnożeniu (przechowuje bazę systemu)

#schemat Hornera
HORNER:
movb bufor(,%esi,1), %bl # odczyt znaku z bufora
```

```
# sprawdzenie czy to koniec znaków w buforze
cmpb $'\n',%bl
je KONIEC

# obliczenie wartości z kodu ASCII
subb $'0',%bl

# przygotownie do mnożenia
mov $BAZA_SYSTEMU, %eax

# mnożenie bazy systemu razy obecna wartość
mul %edx
mov %eax, %edx

# przygotowanie do kolejnego mnożenia o wyżej wadze
add %ebx, %edx

#zwiększenie licznika, w celu przygotowana do odczytu kolejnego znaku
inc %esi
jmp HORNER

KONIEC: #w tym miejscu w rejestrze %edx jest wartosc wprowadzonego
_t9:

#poprawne wyjscie z programu
movl $EXIT, %eax
int $SYSCALL
```

Wyświetlanie liczby w rejestrze w postaci dziesiętnej, szesnastkowej oraz dwójkowej przy pomocy schematu Hornera

Poniższy program również korzysta ze schematu Hornera. Użytkownik wprowadza wartość liczby w postaci binarnej, następnie program konwertuje tę wartość do wybranego systemu. Zmienna BAZA_SYSTEMU określa na jaki system zamienić wprowadzoną liczbę.

```
LICZBA_LITER_ALFABETU = 'Z'-'A'-1
ZERO = 0
DUZA_LITERA = 0x20
BREAK_LINE = 2
BAZA_SYSTEMU = 16
WYPELNIENIE_JEDYNKAMI = -1;
.data
#bufor do przechowywania tekstu do szyfrowania
buf_size = 31
text_size: .long 0
bufor: .space buf_size

horner_buf: .long 31
inv_horner_buf: .long 31

.text
```

```
.global _start
_start:
#wczytwanie z klawiatury do bufora ciagu znakow do zakodowania
movl $READ, %eax
movl $STDIN, %ebx
movl $bufor, %ecx
movl $buf_size, %edx
int  $SYSCALL

movl %eax, text_size #liczba wpisanych znakow w text_size
subl $BREAK_LINE, text_size #odjecie entera od liczby znakow

# --------------START - CZYTANIE BIN W ASCII DO REJESTRU EAX
movl $ZERO, %eax #miejsce gdzie bede przechowywac wartosc
movl $WYPELNIENIE_JEDYNKAMI, %esi

CZYTAJ_ZNAK:
incl %esi
movb bufor(%esi,1), %bl

#program naiwnie zaklada, że uzytkownik poprawnie wprowadzi tylko 1 i 0
subb $'0', %bl #wartosc ze znaku ASCII, 1 lub 0
shll $1, %eax #przesuniecie w prawo o 1, zeby zrobic miejsce na nowy bit
orb %bl, %al #zamiana najmłodego bitu jeśli na wejsciu jest '1' (czyt. z bufora)

#sprawdzenie czy koniec bufora
cmp text_size, %esi #czy koniec bufora?
jb CZYTAJ_ZNAK #wroc do petli jesli to nie koniec bufora

# --------------KONIEC - CZYTANIE BIN W ASCII DO REJESTRU EAX
#teraz w rejestrze EAX, powinna byc wartosc tego co wpisal uzytkownik jako kod binarny
_t0: #pulapka dla debuggera

# --------------START - SCHEMAT HORNERA
movl $ZERO, %esi #wyzerowanie licznika od bufora horner_buf

#schemat Hornera
HORNER:
cdq #Convert Double to Quad
movl $BAZA_SYSTEMU, %ebx #będziemy dzielic przez bazę systemu
div %ebx #dzielimy przez baze systemu, wynik w EAX, reszta w EDX
movl %edx, horner_buf(,%esi,1) #zapis reszty
incl %esi
cmpl $ZERO, %eax #czy to juz koniec?
jne HORNER
# --------------STOP - SCHEMAT HORNERA


# --------------START - ZAMIANA WARTOSCI NA KODY ASCII W TABLICY HORNER_BUF

movl %esi, %eax
```

```
movl $ZERO, %esi
#zamiana kazdej wartosci w tablicy horner_buf na znak ASCII
HORNER_TO_ASCII:
movb horner_buf(,%esi,1), %bl
addb $'0', %bl # utworzenie znaku ASCII

cmp $'9', %bl # jak większe od 9 to musi dodac inna wartosc
jbe ZAMIANA_BUFORA
addb $7, %bl # utworzenie znaku ASCII

ZAMIANA_BUFORA:
movb %bl, horner_buf(,%esi,1)
incl %esi
cmp %eax, %esi
jne HORNER_TO_ASCII

# --------------STOP - ZAMIANA WARTOSCI NA KODY ASCII W TABLICY HORNER_BUF

# --------------START - ODWROCENIE BUFORA HORNER_BUF
movl %esi, %eax # %eax wykorzystane do sprawdzenia czy koniec bufora
movl %esi, %edi # wykorzystane do inv_horner_buf
movl $ZERO, %esi #zerowanie licznika który iteruje po horner_buf

INVERT_HORNER_BUF:
movb horner_buf(,%esi,1), %bl
movb %bl, inv_horner_buf(,%edi,1)
incl %esi
decl %edi
cmp %eax, %esi
jne INVERT_HORNER_BUF

incl %esi #dodatkowa cyfra do wyświetlenia
# --------------STOP - ODWROCENIE BUFORA HORNER_BUF

#wyswietlenie z bufora
WYSWIETL:
movl $WRITE, %eax
movl $STDOUT, %ebx
movl $inv_horner_buf, %ecx
movl %esi, %edx
int  $SYSCALL

#dodanie znaku lamania linii w celu lepszej czytelnosci wyniku
movl $'\n', %ecx
movl $1, %edx
int $SYSCALL

#poprawne wyjscie z programu
movl $EXIT, %eax
int  $SYSCALL
```

1.2 Wnioski

Progamy, które korzystają ze schematu Hornera są bardziej uniwersalne. Powyższe programy potrafią konwertować liczby z danego systemu na dowolny o bazie od 1 do 42. Wynika to z faktu, że liczby większe od 9 zostaną zakodowane w postaci kolejnych znaków alfabetu. 10 cyfr + 32 znaki alfabetu dostępne są w tablicy ASCII. Przy próbie konwersji na liczby o wyższej bazie prawdopodobnie program poinformuje użytkownika o naruszeniu ochrony pamięci.

W powyższym przykładzie pojawiła się instrukcja `cdq`. Jest ona odpowiedzialna za zamianę wartości typu `Double WORD` (32-bity) w rejestrze `eax` na wartość `Quad WORD` (64-bity) zawartą w `edx:eax` przez wypełnienie `edx` wartością najbardziej znaczącego bitu w `eax`.

2 Treść ćwiczenia

Zakres ćwiczenia:

- Pojęcie funkcji w programie asemblerowym
- Korzystanie z instrukcji stosowych `POP` oraz `PUSH`
- Korzystanie z instrukcji `CALL` oraz `RET`
- Funkcja rekurencyjna - obliczanie silni

Zrealizowane zadania:

- Program wykorzystujący instukcję `CALL` oraz `RET`
- Modyfikacja programu z poprzednich laboratoriów
- Funkcja rekurencyjna obliczająca silnię

2.1 Przebieg ćwiczenia

Program wykorzystujący instukcję `CALL` oraz `RET`

Poniższy program prezentuje proste użycie funkcji, przesyłanie parametrów oraz zwracanie wyniku. Funkcja opatrzona etykietą `dodaj` realizuje funkcję dodawania 2 argumentów. Argumenty są przesyłane przez stos, wynik zwracany jest poprzez zastąpienie wartości w adresie pierwszego argumentu.

```
_start:
push $10  #pierwszy parametr
push $5 #drugi parametr
call dodaj # wywolanie funkcji

mov $EXIT, %rax
int $SYSCALL

dodaj:
push %rbp #zapis stosu programu głównego
mov %rsp, %rbp # ustawienie nowej ramki miejscu aktualnej pozycji stosu
mov 16(%rbp), %rax # pobranie 1 parametru, czyli 10
mov 24(%rbp), %rbx # pobranie 2 parametru, czyli 5

#proste obliczenia, wynik w rbx
add %rax, %rbx
mov %rbx, 16(%rbp) #zwrocenie wyniku w miejsce pierwszego argumentu

# przywrocenie stosu POPRZEDNIEGO wywołania funkcji
mov %rbp, %rsp
pop %rbp
ret #powrót do adresu instrukcji zapisanej na stosie
```

Funkcja rekurencyjna obliczająca silnię

Kolejny program korzysta z rekurencji w celu obliczenia silni. Rekurencja jest to odwoływanie się do definicji samej siebie. Przesyłany jest jeden argument poprzez stos. Program odkłada na stos kolejne ramki rekurencyjnych wywołań, dopóki wartość w rejestrze `rbx` nie jest równa 1. Następnie wykonywany jest etap powrotów do adresów powrotów, które zostały zapisane na stosie oraz mnożenie liczby w rejestrze. Wynik jest zwracany w rejestrze `rax`.

```
push $5 #przeslanie parametru
call silnia
_t9: #wynik w %rax
#poprawne wyjscie z programu
mov $EXIT, %rax
int $SYSCALL

silnia:
push %rbp #zapis stosu programu głównego
mov %rsp, %rbp # ustawienie nowego stosu w miejscu aktualnej pozycji stosu
mov 16(%rbp), %rbx # pobranie 1 parametru, czyli 5
cmp $1, %rbx # jeśli 1 to koniec, dalej nie ma potrzeby wchodzi w rekurencje
je silnia_ret #do ret
dec %rbx
push %rbx #kolejny argument na stos, zmniejszony o 1

call silnia # rekurencja
# miejsce powrotu ret

# pomnożenie aktualnego parametru razy wynik poprzedniego wywołania
mov 16(%rbp), %rbx
imul %rbx, %rax
# czyli: n * factorial( n - 1 )

jmp koniec  # koniec
silnia_ret:
mov $1, %rax
koniec:
# przywrocenie stosu POPRZEDNIEGO wywołania funkcji
mov %rbp, %rsp
pop %rbp
ret #powrót do adresu instrukcji zapisanej na stosie
```

2.2 Wnioski

Wszystkie programy udało się w całości zrealizować podczas zajęć laboratoryjnych. Dodatkowo stworzono program który pobierał wejscie od użytkownika, wykonywał funkcję dodaj z zadania 1, oraz obliczał silnię, a następnie wyświetlał wynik w postaci dziesiętnej w terminalu. Było to możliwe dzięki modyfikacji poszczególnych programów w taki sposób aby były funkcjami. Jedynym napotkanym problemem okazało się obserwowanie wskaźnika stosu w programie `gdb`, które było nieporęczne, dlatego wykorzystano narzędzie o nazwie `ddd`.

3 Treść ćwiczenia

Zakres ćwiczenia:

- Wstawianie kodu assemblera `inline` w kodzie C
- Korzystanie z funkcji bibliotecznych języka C w kodzie assemblera
- Korzystanie z własnych funkcji napisanych w języku C w kodzie assemblera
- Korzystanie z własnych funkcji napisanych w języku assemblera w kodzie C

Zrealizowane zadania:

- Użycie funkcji `scanf` oraz `printf` w programie assemblerowym
- Użycie własnej funkcji assemblerowej w kodzie C
- Użycie własnej funkcji w języku C w kodzie assemblera
- Wstawka assemblerowe `inline` w kodzie C

3.1 Przebieg ćwiczenia

Użycie funkcji `scanf` oraz `printf` w programie assemblerowym

```
.align 32
.data

format_input: .string "%d"
zmienna: .long 0

format_string:
.asciz "Test printf to jest tekst: %s, a to jest liczba: %d\n"
text: .string "test"
.global main
main:
#pobranie zmiennej
push $zmienna
push $format_input
call scanf

#wypisanie zmiennej
push zmienna
push $text
push $format_string
call printf
call exit
```

Program należało skonsolidować przy pomocy narzędzie `gcc`.

```
gcc lab.s -m32
./a.out
```

Flaga -m32 jest odpowiedzialna za dostosowanie kodu assemblera do architektury 32 bitowej. Ma to znacznie w przypadku korzystania ze stosu.

Użycie własnej funkcji assemblerowej w kodzie C

Przykładowa funkcja napisana w języku assembler. Ma ona za zadanie wypisywać dwa podane parametry, które zostały przesłane jako argumenty poprzez stos.

```
.data
informacja: .ascii "Funkcja z asm\n"
rozmiar_informacja = . - informacja
format_output: .string "Dwa parametry: %d, %d\n"
zmienna: .long 0
.globl  wypisz
.text

wypisz:
push %ebp #zapis stosu programu głównego
mov %esp, %ebp # ustawienie nowego stosu w miejscu aktualnej pozycji stosu
#4n+8 dla 32 bitow
mov 8(%ebp), %eax
mov 12(%ebp), %ebx

push %eax
push %ebx
push $format_output
call printf

mov %ebp, %esp
pop %ebp
ret #powrót do adresu instrukcji zapisanej na stosie
```

Przykład użycia powyższej funkcji w języku C.

```
#include <stdio.h>
void wypisz(int a, int b);

int main() {
  int a = 0;
  int b = 0;
  scanf("%d", &a);
  scanf("%d", &b);
#pobranie wejście od użytkownika

  wypisz(a, b); #wywołanie funkcji assemblerowej
  return 0;
}
```

Program należało skompilować przy użyciu narzędzie gdb z odpowiednimi argumentami.

```
gcc wypisz.c wypisz.s -m32 #kompilacja
./a.out #uruchomienie
```

Użycie własnej funkcji w języku C w kodzie assemblera

Przykład funkcji wypisującej dwa argumenty przy pomocy metody bibliotecznej `printf`

```
void wypisz(int a, int b) {
   printf("testowe dane, a=%d b=%d \n", a, b);
}
```

Przykładowe użycie powyżeszej funkcji w programie w języku C.

```
.data
informacja: .ascii "Funkcja z asm\n"
rozmiar_informacja = . - informacja
.globl main
.text
main:
push $10
push $5
call wypisz
call exit
```

Program należało skompilować przy użyciu narzędzie `gdb` z odpowiednimi argumentami.

```
gcc wypisz.c wypisz.s -m32 #kompilacja
./a.out #uruchomienie
```

Wstawka assemblerowa `inline` w kodzie C

Poniższy kod dodaje dwie liczby. Dzięki użyciu słowa kluczowego `volatile` mamy pewność że kompilator nie spróbuje zoptymalizować napisanej wstawki. Są one mało wygodnym sposobem wprowadzania kodu assemlera w kodzie C.

```
#include <stdio.h>
int main() {
   int x = 0;
  int z = 0;
  int w = 0;
  printf("Dodawanie 2 liczb\n");
  printf("a = ");
  scanf("%d", &x); #pobranie wejścia
  printf("b = ");
  scanf("%d", &z); #pobranie wejscia

  __asm__ __volatile__ ( #deklaracja wstawki
  "addl %%ecx, %%ebx\n" #dodanie jednej liczby do drugiej
  "movl %%ebx, %%edx\n" #przeniesienie wyniku do %edx
  : "=c"(x),"=b"(z),"=d"(w) # w ecx umieszczono x, ebx - z, edx - w
  : "c"(x), "b"(z), "d"(w) #wartosci przeznaczone do usuniecia
  );
  printf("Wynik: %d\n", w);
  return 0;
}
```

4 Treść ćwiczenia

Zakres ćwiczenia:

- Odczyt stanu jednostki zmiennoprzecinkowej
- Wykonywanie podstawowych obliczeń przy pomocy jednostki zmiennoprzecinkowej
- Zapoznanie się z dostępnymi instrukcjami jednostki zmiennoprzecinkowej oraz specyfiką pracy

Zrealizowane zadania:

- Odczyt wszystkich wyjątków
- Wywołanie wszystkich możliwych wyjątków
- Obliczanie pierwiastków równania kwadratowego - prosta aplikacja zmiennoprzecinkowa

4.1 Przebieg ćwiczenia

Odczyt wszystkich możliwych wyjątków

Program odczytuje stan rejestru statusu jednostki zmiennoprzecinkowej. 16 bitowy rejestr statusu kooprocesora zawiera informację o 6 możliwych wyjątkach.

- Invalid operation exception (bit 0)
- Denomalized exception (bit 1)
- Zero divide exception (bit 2)
- Overflow exception (bit 3)
- Underflow exception (bit 4)
- Precision exception (bit 5)

```
#wyjatki
invalid: .string "Niedozwolona operacja\n"
denormal: .string "Liczba zdenormalizowna\n"
zero: .string "Dzielenie przez zero\n"
overflow: .string "Przepelnienie\n"
underflow: .string "Underflow\n"
precision: .string "Precision\n"
stack: .string "Stack fault\n"
status: .int 127

show_exceptions:
mov $0, %edi #czyszczenie rejestru przez wprowadzeniem resestru statusu
#zapis rejestru jednostki zmiennoprzecinkowej do rejestru %edi
fstsw status
mov status, %edi
```

```
#sprawdzenie czy wystąpił wyjątek nie poprawnej operacji
invalid_e:
test $1, %edi #sprawdź czy na pierwszym bicie znajduje się '1'
jz denorm_e #jesli zero (nie ma wyjatku), to sprawdź kolejny
push $invalid #jeśli jest to wyślij na stos tekst danego wyjątku
call printf #wywołaj funkcję printf z informację o danym wyjątku

denorm_e:
test $2, %edi #sprawdź czy na drugim bicie znajduje się '1'
jz zero_e
push $denormal
call printf

zero_e:
test $4, %edi
jz overflow_e
push $zero
call printf

overflow_e:
test $8, %edi
jz underflow_e
push $overflow
call printf

underflow_e:
test $16, %edi
jz precision_e
push $underflow
call printf

precision_e:
test $32, %edi
jz clear #jeśli nie ma wyjątku to skocz do końca funkcji
push $precision
call printf
```

Wywołanie wszystkich możliwych wyjątków

Program wywołuje wszystkie 6 wyjątków, które mogą zostać odczytane dzięki powyższemu programowi.

```
finit
# invalid operation
fld1 #laduje 1.0 na stos
fchs #zmienia znak na ujemny
fsqrt #zglasza błąd niepoprawnej operacji

# dzielenie przez zero
fld1 #daje 1.0 na stos
fldz # daje 0.0 na stos
```

```
fdivrp #zglasza blad dzielenie przez 0

# underflow i precision
pushl $32829
fild (%esp) # daje na stos
fld1 #daje 1.0 na stos
fscale # potega 2

# precision i overflow
pushl $16384
fild (%esp) # daje 16384 na stos
pushl $16384
fild (%esp) # daje 16384 na stos
fscale #zglasza blad precyzji oraz nadmiaru

# liczba zdenormalizowna
pushl $0x0000001
fld (%esp)
fldz #laduje +0.0 na stos
faddp
```

Instrukcja `finit` inicjalizuje jednostkę zmiennoprzecinkową oraz resetuje jej stan poprzez maskowanie wyjątków. Wyjątki mogą również zostać skasowane dzięki instrukcji `fclex`.

Obliczanie pierwiastków równania kwadratowego

Przykład programu obliczającego pierwiastki równania kwadratowego. Programu nie udało się zrealizować na zajęciach i został on napisany w domu.

```
mov 8(%ebp), %eax  #argument a
mov 12(%ebp), %ebx # argument b
mov 16(%ebp), %ecx  # argument c

finit
flds 12(%ebp) #wyslanie argumentu b na stos
fmul 12(%ebp) #pomnożenie argumentu b*b
# wynik w st(0) b*b

flds const4 # 4 w st(1)
fmul 8(%ebp) #pomnożenie 4 * argument a
#wynik w st(1) 4*b
#wynik w %st(0) 4*b*c
fmul 16(%ebp) #pomnożenie ()argument a * 4) razy argument c

fsub %st(0), %st(1) #odjęcie od ()b*b) - ()4*a*c)
#delta w %st0
_delta:
fsqrt #pierwiastek(delta) w st(0)

flds const2
fmul 8(%ebp)
# 2 * a w st(0)
```

```
flds 12(%ebp)
fmul neg; #pomnożenie *-1 w celu odwrócenia znaku
# -b w st(0)

# fsub %st(2), %st(0) #jeden pierwiastek
# fadd %st(2), %st(0) #drugi pierwiastek
fdiv %st(1), %st(0)

#odłożenie na stos wyniku w st(0)
subl $8, %esp
fstpl (%esp)
pushl $format
call printf #wyświetlenie pierwiastka równania kwadratowego
# -b - sqrt(delta)
```

4.2 Wnioski

Obsługa jednostki zmiennoprzecinkowej znacząco różni się od obsługi rejestrów ogólnego przeznaczenia. Pisanie programów, które w optymalny sposób korzystają ze stosu x87 wydaje się być dość skomplikowanym zadaniem. Większość instrukcji oferuje możliwość ściagnięcia ze stosu wartości po przez dodanie przyrostka `p` (`pop`) do instrukcji.

Literatura

[1] `http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium/Dokumentacja/Programowanie/Linux-asm-lab-2015.pdf`, informacje wstępne dla studentów Politechniki Wrocławskiej na temat `gdb` oraz podstawowe kody źródłowe programów assemblerowych.

[2] `https://pl.wikipedia.org/wiki/Rekurencja`, definicja rekurencji.

[3] J. Biernat, *Profesjonalne przygotowanie publikacji*, materiały konferencyjne X Krajowej Konferencji KOWBAN, str. 401–408, Wyd. WTN, Wrocław, 2003.

[4] `http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium/Dokumentacja/Linux/ddd.pdf`, opis narzędzie `ddd`, graficzna wersja `gdb` *Linux/x86*.

[5] `http://www.linuxassembly.org`, witryna internetowa z informacjami dla programistów asemblera dla platformy *Linux/x86*.

[6] `http://www.x86-64.org/documentation/assembly.html`, witryna internetowa z informacjami dla programistów asemblera dla platformy *Linux/x86*.

[7] `https://en.wikipedia.org/wiki/IEEE_floating_point`, informacje na temat zmiennego przecinka.

[8] `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html`, informacje na temat kodu assemblera w kodzie C.