Wrocław, 5 czerwca 2019

Emilia Pawlaszek
241279
PN 9 TP

Sprawozdanie z Laboratorium przedmiotu „Architektura Komputerów”

Rok akad. 2018/2019, kierunek: INF

Prowadząca:
Aleksandra Postawka

Spis treści

1 Zajęcia wprowadzające

1.1 Cel ćwiczenia

Celem laboratorium wprowadzającego było poznanie technik tworzenia i uruchamiania programów napisanych w języku asemblera oraz obsługa debugger'a gdb. Na zajęciach nauczyliśmy się również podstawowych funkcjonalności konsoli Linuxa i poznaliśmy potrzebne komendy.

1.2 Przebieg ćwiczenia

W trakcie zajęć poznaliśmy podstawowe polecenia używane do obsługi konsoli takie jak:

- `mkdir` - stworzenie katalogu
- `cd` - przejście do katalogu
- `touch` - stworzenie pliku
- `pwd` - wyświetlenie obecnej ścieżki
- `ls` - wyświetlenie zawartości katalogu

Następnie utworzyliśmy pierwszy program „Hello, world!" oraz plik Makefile aby za pomocą prostej komendy make można było kompilować i konsolidować program.
Kolejnym zadaniem było napisanie programu wczytującego tekst i zamieniającego wielkie litery na małe i małe na duże.

1.2.1 Program „Hello, world!"

Na początku programu znajduje się sekcja danych `.data`. Zawiera ona definicje nazw symbolicznych oraz deklaracje zmiennych i buforów. `buf` to bufor w którym zapisany jest nasz napis, a w `buf_len` zapisana jest jego długość.

Sekcja `.text` zawiera algorytm programu. Funkcja `movq` kopiuje do danych rejestrów nazwę i argumenty funkcji. `syscall` służy do wywoływania funkcji.

```
.data
SYSREAD = 0
SYSWRITE = 1
SYSEXIT = 60
STDOUT = 1
STDIN = 0
EXIT_SUCCESS = 0

buf: .ascii "Hello, world!\n"
buf_len = .-buf

.text
.globl _start

_start:

movq $SYSWRITE, %rax
movq $STDOUT, %rdi
movq $buf, %rsi
movq $buf_len, %rdx
syscall

movq $SYSEXIT, %rax
movq $EXIT_SUCCESS, %rdi
syscall
```

1.2.2 Kompilacja programu i uruchomienie programu

Plik Makefile zawiera linijkę z konsolidacją i kompilacją programu. Aby wykonać kompilację należy użyć komendy `make`. Do uruchomienia programu służy komenda `./<nazwa pliku>`.

```
plik: plik.o
      ld -o plik plik.o
plik.o:  plik.s
      as -o plik.o plik.s
```

1.2.3 Debugowanie programu w gdb

Aby przeprowadzić debugowanie programu za pomocą gdb należy skompilować program w poniższy sposób:
`gcc -g -o plik plik.s`
Wtedy w kodzie programu należy zamienić:
`.globl` na `.global` oraz: `_start` na `main`.

W gdb dostępne są następujące opcje:

- ustawienie breakpointa `b [adres]`
- uruchomienie programu `run`
- krokowe działanie programu (`s`)
- kontynuowanie normalnej pracy programu (`c`).
- wyświetlenie zawartości wszystkich rejestrów (`info registers`)
- wyświetlenie zawartości pojedynczego rejestru (`p/d $<nazwa rejestru>`)
- wyświetlenie zawartości danej zmiennej `&<nazwa zmiennej>`
- wyjście z debuggera `quit`

1.2.4 Program zamieniający wielkość liter

Funkcja `SYSREAD` służy do wczytania tekstu z klawiatury,
a `STDIN` to standardowy strumień wejściowy.
Zmienna `BUFLEN` zawiera długość bufora z tekstem.
W sekcji .bss zawarte są niezainicjowane dane, w tym przypadku są to bufory, do których zostanie zapisany wprowadzony tekst.
Na samym początku zostaje zmniejszona o 1 zawartość rejestru `rax`, aby pominąć znak końca linii.
Następnie zapisujemy zero do rejestru `rdi` aby mógł posłużyć nam jako licznik.

W pętli `zamien_wielkosc_liter` zawartość bufora jest kopiowana po jednym bajcie do rejestru `bh`. Następnie zostaje wczytana do rejestru `bl` wartość liczby `0x20`.
Później za pomocą działania `xor` zamieniana jest wielkość litery. Dzieje się tak dlatego, że liczba `0x20` posiada tylko jeden bit 1 na miejscu, w którym różnią się liczby wielkie od małych.
Następnie powstały znak zostaje zapisany do `textout`, licznik pętli zostaje zwiększony i porównany za pomocą funkcji `cmp` z długością bufora wejściowego. Jeśli licznik jest mniejszy to powracamy do pętli (wykorzystanie skoku warunkowego - `jl` - jump if less).

Po wykonaniu się pętli do bufora dodawany jest znak końca linii, po czym funkcja SYSWRITE drukuje do strumienia `STDOUT` zawartość bufora `textout`.

```
.data
STDIN = 0
STDOUT = 1
SYSWRITE = 1
SYSREAD = 0
SYSEXIT = 60
EXIT_SUCCESS = 0
BUFLEN = 512

.bss
.comm textin, 512
.comm textout, 512

.text
.global main
main:
movq $SYSREAD, %rax
movq $STDIN, %rdi
movq $textin, %rsi
movq $BUFLEN, %rdx
syscall

dec %rax
movq $0, %rdi

zamien_wielkosc_liter:
movb textin(, %rdi, 1), %bh
movb $0x20, %bl
xor %bh, %bl
movb %bl, textout(,%rdi, 1)
inc %rdi
cmp %rax, %rdi
jl zamien_wielkosc_liter

movb $'\n', textout(, %rdi, 1)
movq $SYSWRITE, %rax
movq $STDOUT, %rdi
movq $textout, %rsi
movq $BUFLEN, %rdx
syscall
movq $SYSEXIT, %rax
movq $EXIT_SUCCESS, %rdi
syscall
```

1.3 Podsumowanie i wnioski

Programy po kompilacji i uruchomieniu działały tak jak powinny. Podczas pisania kodu w języku Asemblera trzeba mieć na szczególnej uwadze aby nie nadpisywać wartości rejestrów. Dobrą praktyką jest umieszczanie licznych komentarzy w kodzie.

2 Pętle, podstawowe operacje logiczne i arytmetyczne

2.1 Cel ćwiczenia

Celem pierwszych ćwiczeń było zapoznanie z działaniem pętli, operacji logicznych i arytmetycznych oraz praktyczne ich wykorzystanie w różnych programach.

2.2 Przebieg ćwiczenia

Na zajęciach mieliśmy do napisania dwa programy:

1. Zaimplementowanie algorytmu Euklidesa, czyli wyznaczenie największego wspólnego dzielnika dwóch liczb za pomocą operacji dzielenia. Wynik miał być widoczny w gdb.
2. Program, który zamienia wpisany ciąg znaków w reprezentacji trójkowej na ciąg znaków w reprezentacji siódemkowej. Poszczególne etapy zadania:
   - wczytanie liczby z stdin w formacie tekstowym w reprezentacji trójkowej
   - sprawdzenie poprawności wpisanego ciągu znaków
   - zamiana na liczbę w rejestrze (podgląd w GDB)
   - konwersja na system siódemkowy w formacie tekstowym i wypisanie wyniku na ekran

2.2.1 Algorytm Euklidesa

W `DZIELNA_A` i `DZIELNIK_B` umieszczone są stałe wartości - są to liczby dla których chcemy znaleźć największy wspólny dzielnik. Wartości tych liczb umieszczamy następnie za pomocą funkcji movq w rejestrach kolejno `rax` i rbx.

W pętli `poczatek` zerujemy rejestr `rdx`, a następnie porównujemy nasze dwie liczby. Jeśli są sobie równe to za pomocą skoku `je` (skok, jeśli równe) przechodzimy do etykiety z końcem programu. Dzieje się tak dlatego, ponieważ największym wspólnym dzielnikiem dwóch takich samych liczb jest wartość każdej z tych liczb.

Jeśli liczby nie są sobie równe to przechodzimy do części programu z etykietą `algorytm`. Tam dzielimy zawartość rejestru `rax` przez zawartość rejestru `rbx`, wynik tego działania zapisywany jest w rejestrze `rax`, a reszta w `rdx`.
W następnym kroku dzielnik staje się dzielną, czyli z rejestru `rbx` przechodzi do rejestru `rax`. W kolejnej linijce reszta staje się dzielnikiem, czyli z rejestru `rdx` przechodzi do rejestru `rbx`.
Następnie sprawdzamy czy reszta jest już równa zero, jeśli tak to największy wpólny dzielnik naszych dwóch liczb znajduje się w rejestrze `rax` - jest to aktualny dzielnik. W przeciwnym przypadku, czyli jeśli reszta jest różna od zera, za pomocą skoku `jne` (skok, jeśli nierówne) przechodzimy do etykiety `początek`.

Wynik możemy zobaczyć w gdb za pomocą komendy `print $rax`.

```
.data
SYSWRITE = 1
SYSEXIT = 60
STDOUT = 1
EXIT_SUCCESS = 0
DZIELNA_A = 14
DZIELNIK_B = 35

.text
.globl _start

_start:
movq $DZIELNA_A, %rax
movq $DZIELNIK_B, %rbx

poczatek:
movq $0, %rdx
cmp %rax, %rbx
je koniec

algorytm:
div %rbx
movq %rbx, %rax
movq %rdx, %rbx
cmp $0, %rdx
jne poczatek

koniec:
movq $SYSEXIT, %rax
movq $EXIT_SUCCESS, %rdi
syscall
```

2.2.2 Zamiana systemów liczbowych

Pierwsza część programu odpowiedzialna jest za wyświetlenie napisu początkowego. Kolejna za wczytanie liczby od użytkownika.
w pętli następuje sprawdzenie czy podana liczba jest w systemie trójkowym. Zostają porównane kody ASCII kolejnych cyfr.
Następnie odkodowujemy naszą cyfrę za pomocą odjęcia '0' od znaku ASCII danej cyfry.
W `first` porównujemy obecny indeks z zerem, jeśli nie jest tyle równy to przechodzimy do `second`. Tam naszą liczbę podnosimy do kolejnej potęgi.
Następnie mnożymy wartość liczby przez mnożnik i dodajemy do całej wartości liczby.
Następnie zwiększamy indeks o 1, sprawdzamy czy wczytaliśmy całą liczbę, jeśli nie to wczytujemy kolejną pozycję.

```
msg: .ascii "Wpisz liczbe w sys 3\n"
msg_len = . - msg
endline: .ascii "\n"
endline_len = . - endline
.bss
#bufory
.text
.globl _start
_start:
movq $SYSWRITE, %rax
movq $STDOUT, %rdi
movq $msg, %rsi
movq $msg_len, %rdx
syscall

movq $SYSREAD,  %rax
movq $STDIN, %rdi
movq $firststring, %rsi
movq $BUFLEN, %rdx
syscall

movq $3, %rcx
movq $0, %rbp
movq %rax, %rsp

petla:
movb firststring(,%rdi,1), %bl
cmp $0x19, %bl
jl dalej
cmp $'0', %bl
jl quit
cmp $'2', %bl
jg quit

sub $48, %bl
movq $1, %rax
movq %rdi, %rsi

jmp first
second:
dec %rsi
mul %rcx
first:
cmp $0, %rsi
jne second
```

```
mul %bl
add %rax, %rbp
dalej:
inc %rdi
cmp %rsp, %rdi
jl petla

zmiana_systemu:
movq %rbp, %rax
movq $0, %rsi
movq $7, %rcx

schemat_hornera:
movq $0, %rdx
div %rcx
add $48, %rdx
movb %dl, secondstring(, %rsi, 1)
inc %rsi
cmp $0, %rax
jne schemat_hornera

movq $0, %rdi
odwrocenie:
movb secondstring(, %rdi, 1), %bh
movb %bh, %bl
movb %bl, thirdstring(, %rdi,1)
inc %rdi
cmp $0, %rax
jne odwrocenie

movq $SYSWRITE, %rax
movq $STDOUT, %rdi
movq $thirdstring, %rsi
movq $BUFLEN, %rdx
syscall

movq $SYSEXIT, %rax
movq $EXIT_SUCCESS, %rdi programu
syscall
```

Następnie następuje zamiana systemu na siódemkowy. W rejestrze `rcx` umieszczamy podstawę naszego systemu.
W `schemat_hornera` kodujemy powstałą cyfrę na znak ASCII i dodajemy kolejne znaki do zmienionego ciągu.
Następnie następuje nieudana próba odwrócenia ciągu. Ten fragment kodu jest zły.

Na koniec zostaje załadowanie polecenie odczytu dla systemu i polecenie zakończenia programu.

2.3 Podsumowanie i wnioski

Pierwszy program działał poprawnie. Drugi program źle oblicza wartość liczby (w systemie dziesiętnym), przez co cały działa niepoprawnie. Przy złożonych obliczeniach matematycznych warto rozpisać sobie wszystko na kartce. W trakcie zajęć przydała się wiedza z pierwszej części kursu „Architektura Komputerów”.

3 Utrwalenie umiejętności tworzenia prostych konstrukcji programowych

3.1 Cel ćwiczenia

Celem ćwiczenia było zapoznanie się z obsługą plików w programach w języku Asemblera. Aby odczytać lub zapisać dane do pliku trzeba wykorzystać wywołania systemowe otwierającymi plik do odczytu i/lub zapisu. Podczas otwierania pliku należy zaznaczyć co zamierzamy z nim zrobić – plik możemy otworzyć tylko do odczytu, do zapisu, do zapisu i odczytu itp. W przypadku gdy spróbujemy otworzyć nieistniejący plik do zapisu, plik ten może zostać utworzony. Należy wtedy sprecyzować z jakimi prawami dostępu ma on zostać utworzony.

3.2 Przebieg ćwiczenia

Podczas zajęć mieliśmy napisać jeden, duży program, który miał zawierać w sobie wiele funkcjonalności takich jak wczytywanie i zapis do pliku oraz zamiana systemów liczbowych. Z tego powodu podczas opisywania programu pominę kod, który jest nieistotny i podzielę pozostawione fragmenty na klika części aby ich opis był czytelniejszy.

Kolejne etapy wykonania programu to:

1. Wczytanie z dwóch plików dwóch „dużych” liczb w ASCII zapisanych w reprezentacji szesnastkowej.
2. Zamiana na poprawny zapis w pamięci (konwencja Little Endian)
3. Wykonanie dodawania z użyciem rejestrów 64-bitowych i flagi CF.
4. Zamiana wyniku na zapis tekstowy w reprezentacji ósemkowej i zapis do pliku.

3.2.1 Wczytywanie z plików, dodawanie i konwersje

Sekcja `.data` i sekcja `.bss`

W sekcji `.data` jedyną zmianą są zmienne `FREAD` i `FWRITE`, służą one do obsługi plików.
Nazwy plików muszą się kończyć zerowym bajtem. W sekcji `.bss` zadeklarowane są bufory zawierające znaki odczytane z pliku (pierwsze trzy bufory) oraz bufory zawierające wartości kolejnych bajtów odczytanych liczb(kolejne trzy bufory).

```
.data
FREAD = 0
FWRITE = 1

fileWithFirstNumber: .ascii "first.txt\0"
fileWithSecondNumber: .ascii "second.txt\0"
fileWithOutNumber: .ascii "out.txt\0"

.bss
.comm firstIn, 1024
.comm secondIn, 1024
.comm out, 1024

.comm firstValue, 1024
.comm secondValue, 1024
.comm outValue, 1024
```

Wczytanie pierwszego ciągu znaków

Rejestr `r8` to licznik.
Zerujemy rejestr `al`, aby móc później zapisać do niego wartość pierwszej liczby.
W pętli `loop1_0` wypełniamy bufor `firstValue` (bufor przeznaczony na wartość pierwszej liczby) zerami.

Następnie otwieramy pierwszy plik. Do rejestru `rax` zostaje zapisany numer wywołania systemowego, do `rdi` nazwa pliku, do `rsi` sposób otwarcia pliku (w tym przypadku `FREAD` - tylko do odczytu), a do `rdx` prawa dostępu do pliku.
Instrukcją `movq %rax , %r10` zostaje przypisany identyfikator otwartego pliku do rejestru `r10`.

Następnie następuje odczyt z pliku do bufora. W rejestrze `r10` mamy zapisany identyfikator otwartego pliku. Instrukcją `movq %rax , %r8` zapisujemy ilość odczytanych bajtów do rejestru `r8`.

Ostatnim etapem w tej części kodu jest zamknięcie pliku.

```
movq $1024, %r8
movb $0, %al

loop1_0:
dec %r8
movb %al, firstValue(, %r8, 1)
cmp $0, %r8
jg loop1_0

movq $SYSOPEN, %rax
movq $fileWithFirstNumber, %rdi
movq $FREAD, %rsi
movq $0, %rdx
syscall
movq %rax, %r10

movq $SYSREAD, %rax
movq %r10, %rdi
movq $firstIn, %rsi
movq $1024, %rdx
syscall
movq %rax, %r8

movq $SYSCLOSE, %rax
movq %r10, %rdi
movq $0, %rsi
movq $0, %rdx
syscall
```

Dekodowanie wartości pierwszego ciągu

Najpierw następuje pominięcie znaku końca linii (w rejestrze `r8` znajduje się liczba odczytanych bitów). Później tworzymy licznik, który działa od końca.

W pętli `loop1_1` po zmniejszeniu odpowiednich rejestrów wykonuje się dekodowanie pierwszych czterech bitów, a następnie zamiana na wartość. Cyfrę z litery odkodowujemy poprzez odjęcie liczby 55 od jej kodu ASCII.

W pętli loop1_2 następuje sprawdzenie czy zostały odczytane wszystkie cyfry z bufora firstIn.
Jeśli tak się nie stało to przechodzimy do dekodowania kolejnych 4 bitów. Dzieje się to analogicznie jak wcześniej.

W pętli loop1_3 obliczona zostaje wartość odkodowanej części cyfry poprzez pomnożenie jej przez 16 i zostaje dodana do obecnej liczby w buforze. Następnie odkodowany bajt zostaje zapisany do bufora. Następuje skok na początek pętli jeśli nie została jeszcze odczytana całość bufora wejściowego.

```
dec %r8
movq $1024, %r9

loop1_1:
dec %r8
dec %r9

movb firstIn (, %r8, 1), %al
cmp $'A', %al
jge letter1_0

sub $'0', %al
jmp loop1_2

letter1_0:
sub $55, %al

loop1_2:
cmp $0, %r8
jle loop1_4

movb %al, %bl
dec %r8
movb firstIn(, %r8, 1), %al

cmp $'A', %al
jge letter1_1

sub $'0', %al
jmp loop1_3

letter1_1:
sub $55, %al

loop1_3:
movb $16, %cl
mul %cl
add %bl, %al

loop1_4:
movb %al, firstValue(, %r9, 1)

cmp $0, %r8
jg loop1_1
```

Po tej części programu zostaje wczytany i odkodowany drugi ciąg znaków. Dzieje się to tak samo jak w przypadku pierwszego ciągu. Ten kod zostanie pominięty w sprawozdaniu.

Dodanie obydwóch liczb

Funkcja `clc` ustawia ustawia flagę przeniesienia na 0. Następnie za pomocą `pushfq` rejestr flagowy zostaje umieszczony na stosie. W rejestrze `r8` znajduje się licznik pętli.

w pętli `loop3` następuje odczytanie obydwóch wartości. Następnie za pomocą `popfq` zostaje pobrana zawartość rejestru flagowego ze stosu. Później za pomocą instrukcji `adc` zostają dodane do siebie obydwie wartości wraz z przeniesieniem.
Otrzymana wartość zostaje zapisana do bufora `outValue` i następuje powrót na początek pętli aż do momentu odczytania wszystkich bajtów.

```
clc
pushfq
movq $1024, %r8

loop3:
movb firstValue(, %r8, 1), %al
movb secondValue(, %r8, 1), %bl
popfq
adc %bl, %al

pushfq
movb %al, outValue(, %r8, 1)
dec %r8
cmp $0, %r8
jg loop3
```

Konwersja na system ósemkowy

W tej części skorzystamy z zależności baz skojarzonych aby przekonwertować otrzymaną sumę na system ósemkowy. Program pobiera w odpowiedniej kolejności 3 kolejne bajty z bufora wartości.
Następnie zawartość rejestru `rax` za pomocą instrukcji `shl` zostaje przesunięta o 8 bitów w lewo. Później zostaje pobrany w ten sam sposób drugi i trzeci bajt.
Jeśli sprawdzimy teraz zawartość rejestru `rax` w gdb to będzie ona wyglądała w taki sposób: 0x ab cd ef.

Pobraliśmy w ten sposób 24 bity - jest to 6 znaków w systemie szesnastkowym i 8 znaków w systemie ósemkowym.
Tworzymy licznik dla pętli (rejestr `r10`) aby odczytywać 8 znków z liczby 3 bajtowej.
W pętli `loop5` zapisujemy pierwszy bajt liczby do rejestru `bl`, a następnie wyłuskujemy 3 najmniej znaczące bity i usuwamy resztę bitów. Te bity są wartością liczby w systemie ósemkowym.
Następnie zamieniamy powstałą cyfrę na ASCII i zapisujemy ją do bufora `out`.
Przesuwamy się w rejestrze `rax` o 3 bity w prawo, czyli pozbywamy się 3 odkodowanych bitów.
Po sprawdzeniu czy została odkodowana cała liczba przechodzimy na początek pętli lub na koniec programu.

```
movq $1023, %r8
movq $1022, %r9

loop4:
movq $0, %rax
sub $2, %r8
movb outValue(, %r8, 1), %al

shl $8, %rax
inc %r8
movb outValue(, %r8, 1), %al
shl $8, %rax
inc %r8
movb outValue(, %r8, 1), %al
sub $3, %r8

movq $8, %r10
loop5:
movb %al, %bl
and $7, %bl
add $'0', %bl
movb %bl, out(, %r9,1)
shr $3, %rax
dec %r9
dec %r10
cmp $0, %r10
jg loop5

cmp $0, %r8
jg loop4
```

Zapisanie wyniku

W ostatniej części programu zostaje otworzony plik do zapisu otrzymanego wyniku.
Następnie bufor `out` zostaje zapisany do pliku wyjściowego. Po tym plik zostaje zamknięty i następuje koniec programu.

```
movq $SYSOPEN, %rax
movq $fileWithOutNumber, %rdi
movq $FWRITE, %rsi
movq $0, %rdx
syscall
movq %rax, %r8

movq $1024, %r9
movb $0x0A, out(, %r9, 1)
movq $SYSWRITE, %rax
movq %r8, %rdi
movq $out, %rsi
movq $1025, %rdx
syscall

movq $SYSCLOSE, %rax
movq %r8, %rdi
movq $0, %rsi
movq $0, %rdx
syscall

movq $SYSEXIT, %rax
movq $EXIT_SUCCESS, %rdi
sysca
```

3.3 Podsumowanie i wnioski

Podczas tych zajęć skorzystaliśmy z użytecznego uproszczenia podczas zamiany systemów. Tym uproszczeniem jest użycie baz skojarzonych. Znacznie przyśpiesza i ułatwia to konwersję między systemami. W trakcie zapisu liczb do pamięci należy pamiętać o stosowanej konwencji Little Endian, w której liczby należy zapisywać od najmłodszego bitu.

4 Stos i funkcje

4.1 Cel ćwiczenia

Zapoznanie się z działaniem stosu w procesorach z rodziny x86 oraz wykorzystanie w praktyce funkcji rekurencyjnych.

Stos to struktura danych, w której dane umieszczane są na wierzchołku stosu i również z wierzchołka są pobierane. Rozkaz `push REJESTR` odkłada zawartość rejestru na stos, a `pop REJESTR` ściąga zawartość ostatniego rejestru ze stosu. Rejestr `rsp` przechowuje wskaźnik na ostatni element stosu. Jeżeli zwiększymy zawartość tego rejestru o 8, to usuniemy ze stosu ostatnią wartość.

Funkcje w języku Asemblera wykonuje się za pomocą instrukcji `call` oraz `ret`. Rozkaz `call` odkłada na stos adres z którego nastąpiło wywołanie funkcji, a następnie wykonuje skok do nowej etykiety. Rozkaz `ret` powraca pod ten adres i ściąga go ze stosu.

4.2 Przebieg ćwiczenia

Podczas zajęć mieliśmy do napisania 3 funkcje:

1. Funkcja obliczająca najmniejszą wspólną wielokrotność dwóch liczb.
2. Funkcja rekurencyjna:
$$\begin{cases} x_i = -2 * x_{i-1} + 3 * x_{i-2} \\ x_0 = -1 \\ x_1 = 2 \end{cases}$$
w której argumenty i wynik przekazywane są przez rejestry.
3. Funkcja rekurencyjna opisana tym samym wzorem co powyżej, ale z tą zmianą, że argumenty i wynik przekazywane są przez stos.

4.2.1 Najmniejsza wspólna wielokrotność

Obydwie liczby mamy zapisane w stałych. Następnie wartości tych liczb przenosimy do rejestrów i za pomocą instrukcji `push` umieszczamy te rejestry na stosie.
Instrukcja `call` odpowiedzialna jest za wywołanie funkcji.

W funkcji `oblicz` odkładamy na stos poprzednią wartość rejestru bazowego, następnie pobieramy zawartość rejestru `rsp`, czyli wskaźnika na ostatni element na stosie.
Później za pomocą instrukcji `sub` odejmujemy od rejestru `rsp` osiem, czyli przesuwamy się o jedną komórkę w górze na stosie.
Za pomocą takich „przesunięć" możemy pobierać dowolne argumenty ze stosu tak jak jest to zaimplementowane w kodzie.

W kolejnym etapie mnożymy przez siebie nasze dwie liczby, wynik przechowujemy w rejestrze `r10`.
Następnie wykorzystany zostaje Algorytm Euklidesa (napisany na wcześniejszych zajęciach). Po jego wykonaniu w rejestrze `rax` znajduje się NWD dwóch liczb, który jest w następnym etapie wykorzystany jako dzielnik uprzednio wymnożonej liczby. W ten sposób otrzymujemy NWW, która znajduje się w rejestrze `rax`.

```
.data
PIERWSZA = 108
DRUGA = 72
.text
.globl main
main:
movq $PIERWSZA, %r8
movq $DRUGA, %r9

push %r8
push %r9
call oblicz

oblicz:
push %rbp
movq %rsp, %rbp
sub $8, %rsp

movq 24(%rbp), %rax
movq 16(%rbp), %rbx

# Kod funkcji
mul %rbx
movq %rax, %r10

movq 24(%rbp), %rax
movq 16(%rbp), %rbx

# Algorytm Euklides
# nwd w rax
movq %rax, %r11
movq %r10, %rax
div %r11

koniec:
movq %rbp, %rsp
pop %rbp
ret
```

4.2.2 Rekurencja rejestry

Najpierw następuje sprawdzenie czy obecny wyraz ciągu nie jest zerowym lub pierwszym. Następnie wyraz ciągu zostaje zmniejszony (n = n-1). Funkcję wywołujemy dwa razy, a po każdym tym procesie wynik jest odejmowany od rejestru `rcx` (czyli naszego ostatecznego wyniku).

Później analogicznie wywołujemy funkcje dla n = n-2, z tą różnicą, że tym razem wynik cząstkowy dodajemy do ostatecznego wyniku.

```
.data
NUMER = 6
ZEROWY_WYRAZ = -1
PIERWSZY_WYRAZ = 2

.text
.globl main
main:
movq $NUMER, %r8
movq %r8, %r9
call funkcja
jmp koniec

funkcja:
movq %r9, %rax
cmp $1, %rax
jl zerowy
je pierwszy
movq $0, %rcx

dec %rax
call funkcja
sub %rbx, %rcx
call funkcja
sub %rbx, %rcx

dec %rax
call funkcja
add %rbx, %rcx
call funkcja
add %rbx, %rcx
call funkcja
add %rbx, %rcx

movq %rcx, %rbx
call funkcja

zerowy:
movq $ZEROWY_WYRAZ, %rbx
ret
pierwszy:
movq $PIERWSZY_WYRAZ, %rbx
ret
```

4.2.3 Rekurencja stos

Ilość wyrazów ciągu (`NUMER`) zostaje umieszczona na stosie i zostaje wywołana funkcja. W ciele funkcji umieszczamy na stosie poprzednią wartość rejestru bazowego, pobieramy wskaźnik na ostatni element stosu do rejestru bazowego i zwiększamy wskaźnik stosu o jedną pozycję. Następnie pobieramy zawartość drugiego elementu stosu, czyli numeru wyrazu ciągu) i zapisujemy go do rejestru `rax`.
Analogicznie jak w programie poprzednim sprawdzamy czy wyraz ciągu nie jest zerowy albo pierwszy.

Za każdym razem przed wywołaniem funkcji umieszczamy na stosie wynik i numer obecnego wyrazu ciągu. Po wywołaniu funkcji ściągamy te wartości ze stosu.

```
.text
.globl main
main:
movq $NUMER, %r8
push %r8
call funkcja
add $8, %rsp

funkcja:
push %rbp
movq %rsp, %rbp
sub $8, %rsp
movq 16(%rbp), %rax
cmp $1, %rax
jl zerowy
je pierwszy
movq $0, %rcx

dec %rax
push %rcx
push %rax
call funkcja
pop %rax
pop %rcx
sub %rbx, %rcx

push %rcx
push %rax
call funkcja
pop %rax
pop %rcx
sub %rbx, %rcx

dec %rax
push %rcx
push %rax
call funkcja
pop %rax
pop %rcx
add %rbx, %rcx

push %rcx
push %rax
call funkcja
pop %rax
pop %rcx
add %rbx, %rcx
```

Na koniec zwracamy wyliczoną wartość do rejestru `rbx`. Zapisujemy rejestr bazowy do wskaźnika szczytu stosu, ściągamy ze stosu ostatni element i następuje powrót do miejsca wywołania funkcji.

```
push %rcx
push %rax
call funkcja
pop %rax
pop %rcx
add %rbx, %rcx

movq %rcx, %rbx
movq %rbp, %rsp
pop %rbp
ret

zerowy:
movq $ZEROWY_WYRAZ, %rbx
movq %rbp, %rsp
pop %rbp
ret
pierwszy:
movq $PIERWSZY_WYRAZ, %rbx
movq %rbp, %rsp
pop %rbp
ret
```

4.3 Podsumowanie i wnioski

W pierwszym programie należało wykorzystać kod z poprzednich zajęć, program działał poprawnie.
Program z funkcją rekurencyjną z użyciem rejestrów nie działał prawidłowo. Zapewne było to związane z niepoprawnym zrozumieniem działania stosu.
Trzeci i ostatni program działał połowicznie dobrze. Argumenty otrzymywał przez stos, ale wyniku już przez stos nie zwracał.

5 Łączenie różnych języków programowania w jednym projekcie

5.1 Cel ćwiczenia

Najczęściej kod napisany w języku Asemblera wykorzystujemy z połączeniem kodu napisanego w innych językach programowania. Na przykład w celu optymalizacji obliczeń matematycznych możemy napisać funkcję w Asemblerze i wywołać ją w kodzie napisanym języku C.
Możemy również wywoływać funkcje napisane w C z kodu Asemblerowego. Aby to zrobić należy umieścić argumenty funkcji w kolejnych rejestrach.
Dla argumentów całkowitych: rdi, rsi, rdx, rcx, r8, r9.
Dla argumentów zmiennoprzecinkowych: xmm0 - xmm7 (następuje tu również konieczność przekazania ilości argumentów zmiennoprzecinkowych do rejestru rax).
Jeśli łączymy kody, to aby skompilować i zlinkować je razem, możemy skorzystać z polecenia: gcc plik.s plik.c -o plik -g (-no-pie)

5.2 Przebieg ćwiczenia

Na zajęciach mieliśmy wykonać następujące zadania:

1. Program w języku Asemblera:
   - Wczytanie liczby całkowitoliczbowej x oraz dwóch liczb zmiennoprzecinkowych y oraz z (za pomocą funkcji scanf)
   - Wywołanie funkcji napisanej w języku C:
   $$f(x, y, z) = x^2 + y^2 + z^2$$
   - Wypisanie wyniku (zmiennoprzecinkowego) na ekran za pomocą funkcji printf
2. Program w języku C:
   - Wywołanie funkcji napisanej w języku Asemblera - szyfr Cezara dla cyfr dla zadanego klucza (wywołanie funkcji w ASM, która dla liter w podanym ciągu znaków zastosuje szyfr Cezara, klucz podany w argumencie funkcji)
   $$f(char*, int)$$
   Pozostałe znaki (inne niż cyfry) powinny zostać niezmienione.
3. Wstawka Asemblerowa:
   - szyfr Cezara dla cyfr dla zadanego klucza (dane podane w stałych/ zmiennych)
   $$f(char*, int)$$
   Pozostałe znaki (inne niż cyfry) powinny zostać niezmienione.

5.2.1 Wywołanie w ASM funkcji napisanej w C

Kod ASM:

```
.data
    SYSEXIT = 60
    EXIT_SUCCESS = 0
    decimal: .asciz "%d"
    float: .asciz "%f"
    result: .asciz "%lf\n"
.bss
    .comm num1, 4
    .comm num2, 4
    .comm num3, 4
.text
.global main
main:
    push %r8
    call funkcja
    add $8, %rsp
funkcja:
    push %rbp
    mov %rsp, %rbp

    mov $0, %rax
    mov $decimal, %rdi
    mov $num1, %rsi
    call scanf

    mov $0, %rax
    mov $float, %rdi
    mov $num2, %rsi
    call scanf

    mov $0, %rax
    mov $float, %rdi
    mov $num3, %rsi
    call scanf

wyw_fun:
    mov $2, %rax
    mov num1, %rdi
    movss num2, %xmm0
    movss num3, %xmm1
    call func

    mov $1, %rax
    mov $result, %rdi
    call printf
    movq $SYSEXIT, %rax
    movq $EXIT_SUCCESS, %rdi
    syscall
```

W sekcji `.data` zostały umieszczone zmienne potrzebne do wywołania funkcji w języku C: scanf i printf.
W sekcji `.bss` znajdują się bufory na liczby.
Przed wywołaniem funkcji musimy umieścić cokolwiek na stosie, inaczej program nie zadziała.Po wywołaniu dana wartość jest ściągana ze stosu.
W ciele funkcji najpierw następuje wczytanie liczby całkowitej. Do rejestru `rax` przekazujemy ilość argumentów zmiennoprzecinkowych, pierwszy argument (`rdi`) to format zapisanej liczby, drugi (`rsi`) to adres bufora do którego ma być zapisana liczba. `call scanf` to wywołanie funkcji `scanf` z języka C.
Następnie pobieramy kolejno dwie liczby zmiennoprzecinkowe.
W `wyw_fun` wywołujemy funkcję z C o nazwie `func`. Pod koniec wyświetlamy za pomocą funkcji `printf` wynik na ekran.

Kod C:

```
double func(int x, float y, float z)
{
    return x*x + y*y + z*z;
}
```

5.2.2 Wywołanie w C funkcji napisanej w ASM

W sekcji `.text` następuje deklaracja funkcji którą stworzymy.
Szyfr Cezara działa na zasadzie przesuwania każdej litery w ciągu znaków o dany klucz. Funkcję wywołujemy z dwoma argumentami, są one przechowywane kolejno w rejestrach `rdi` i `rsi`. Zawartość rejestru `rdi` to wskaźnik na pierwszą komórkę stringa, a rejestru `rsi` to długość tego stringa.
Później dla każdego znaku wykonywana jest pętla, która szyfruje dany znak poprzez przesuwanie go w tabeli ASCII.

```
.text
.global szyfr_cezara
.type szyfr_cezara, @function

szyfr_cezara:
    push %rbp
    mov %rsp, %rbp
    mov $0, %rax
    petla:
        mov (%rdi, %rax, 1), %bl
        cmp $'Z', %bl
        jle duze

        male:
            add $4, %bl
            cmp $'z', %bl
            jle dopisz
            sub $26, %bl
            jmp dopisz
        duze:
            add $4, %bl
            cmp $'Z', %bl
            jle dopisz
            sub $26, %bl

        dopisz:
        mov %bl, (%rdi, %rax, 1)

    inc %rax
    cmp %rsi, %rax
    jl petla

    mov %rbp, %rsp
    pop %rbp
ret
```

Najpierw następuje deklaracja funkcji z języka ASM. Można zauważyć tam zwracany typ oraz rodzaj przesyłanych argumentów.
W funkcji `main` zostaje wywołana funkcja zewnętrzna, a wynik zostaje wyświetlony na ekran.

```c
#include <stdio.h>
extern void szyfr_cezara(char * str, int len);

char str[] = "abcabcabc";
int len = 9;

int main(void)
{
    szyfr_cezara(&str, len);
    printf("Rezultat: %s\n", str);
    return 0;
}
```

5.2.3 Wstawka Asemblerowa

W kod napisany w języku C możemy wstawić kod napisany w języku Asemblera. Wystarczy użyć składni `asm()` i wewnątrz tej funkcji umieścić kod opatrzony w apostrofy i znak nowej linii. Każdy rejestr musi być poprzedzony dwoma znakami %%. Możemy korzystać z aliasów kolejnych rejestrów. Aliasy to kolejne cyfry poprzedzone jednym znakiem %.
Wewnątrz znajduje się kod z poprzedniego zadania.
Pod koniec po dwukropku należy podać kolejno: : parametry wyjściowe, : parametry wejściowe, : rejestry z których będziemy korzystać w kodzie Asemblerowym.

```
#include <stdio.h>
char str[] = "abcabcabc";
const int len = 9;

int main(void)
{
    asm(
    "mov $0, %%rbx \n"
    "petla: \n"
    "mov (%0, %%rbx, 1), %%al \n"
    "cmp $'Z', %%al \n"
    "jle duze \n"

   "male: \n"
       "add $4, %%al \n"
            "cmp $'z', %%al \n"
            "jle zapisz \n"
            "sub $26, %%al \n"
            "jmp zapisz \n"

        "duze: \n"
            "add $4, %%al \n"
            "cmp $'Z', %%al \n"
            "jle zapisz \n"
            "sub $26, %%al \n"

    "zapisz: \n"
    "mov %%al, (%0, %%rbx, 1) \n"
    "inc %%rbx \n"
    "cmp len, %%ebx \n"
    "jl petla \n"

    :
    :"r"(&str)
    :"%rax", "%rbx"
    );

    printf("Rezultat: %s\n", str);
    return 0;
```

5.3 Podsumowanie i wnioski

Zarówno w drugim jak i w trzecim programie nie został uwzględniony wymóg dotyczący pozostałych znaków (innych niż cyfry/litery). Zostają przesunięte wszystkie znaki. Używanie funkcji z C w Asemblerze może znacznie ułatwić pisanie kodu. Używanie funkcji napisanych w Asemblerze do obliczeń w języku C może znacząco przyspieszyć te obliczenia.

6 Jednostka zmiennoprzecinkowa(FPU)

6.1 Cel ćwiczenia

Koprocesor arytmetyczny (Floating Point Unit, FPU) to układ scalony wspomagający procesor w obliczeniach zmiennoprzecinkowych i nie tylko.
FPU ma 8 identycznych 80-bitowych rejestrów zmiennoprzecinkowych i 3 rejestry kontrolne (Control Word, Status Word, Tag Word). Rejestry zmiennoprzecinkowe ułożone są w stos. Możliwe jest wstawianie do nich wartości i pobieranie z nich wartości. Na dole stosu zawsze znajduje się rejestr ST(0) i to właśnie do niego wstawiana jest nowa wartość. Jeśli wstawimy kolejną wartość to wartość rejestru ST(0) przesunie się do rejestru ST(1), a nowa wartość zostanie zapisana do ST(0).
Analogicznie będą przesuwane pozostałe rejestry. Przy pobieraniu wartości ze stosu numeracja działa podobnie, ale w drugą stronę – indeksy zmniejszają się.

REJESTRY KONTROLNE:

- `Control Word` - ten rejestr odpowiedzialny jest za ustawienia jednostki FPU.
  Interesujące nas bity Control Word:
  PM (bit 5) Precision Mask
  UM (bit 4) Underflow Mask
  OM (bit 3) Overflow Mask
  ZM (bit 2) Zero divide Mask
  DM (bit 1) Denormalized operand Mask
  IM (bit 0) Invalid operation Mask

- `Status Word` - wskazuje ogólny stan jednostki FPU.
  Na początku bity ustawione są na 0. Bity 6-0 to flagi uaktywnione po detekcji wyjątku. Można je zresetować za pomocą instrukcji finit. Instrukcja ta resetuje wszystkie rejestry i flagi do ich domyślnych wartości.
  Interesujące nas bity Status Word:
  SF (bit 6) Stack Fault exception flag
  P (bit 5) Precision exception flag
  U (bit 4) Underflow exception flag
  O (bit 3) Overflow exception flag
  Z (bit 2) Zero divide exception flag
  D (bit 1) Denormalized exception flag
  I (bit 0) Invalid operation exception flag

- `Tag Word` - zawiera informacje o zawartości każdego z 80-bitowych rejestrów. Podzielony jest na pary bitów.
  Znaczenie każdej z par bitów:
  00 = The register contains a valid non-zero value
  01 = The register contains a value equal to 0
  10 = The register contains a special value (NAN, infinity, or denormal)
  11 = The register is empty

6.2 Przebieg ćwiczenia

Podczas zajęć do wykonania były następujące zadania:

1. Maskowanie wyjątków:
   - Funkcja w ASM sprawdzająca wskazaną flagę (dla określonego wyjątku) w Status Word.
   - Funkcja w ASM modyfikująca ustawienia maskowania wyjątków w Control Word (dla wskazanego wyjątku).
   - Wykazanie różnicy w działaniu dla maskowania i nie maskowania wyjątków.
2. Szereg Taylora dla

   $$e^x$$

   czyli

   $$1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

   - funkcja w ASM.

6.2.1 Maskowanie wyjątków

Kod ASM:

Funkcja `sprawdzWyjatki` odpowiedzialna jest za sprawdzenie wskazanej flagi w Status Word. Instrukcja `fstsw` wczytuje zawartość Status Word do rejestru `ax`. Za pomocą `fwait` program czeka na wykonanie poprzedniej instrukcji. Instrukcja logiczna and czyści starszą część rejestru, ponieważ interesują nas tylko flagi wyjątków. Wynik funkcji znajduje się w rejestrze `rax`.

Funkcja `wyczyscMaski` odpowiedzialna jest za wyczyszczenie masek wyjątków (domyślnie są ustawione). Jeśli je wyczyścimy to możliwe będzie wywołanie wyjątku.
Najpierw wczytujemy zawartość Control Word do rejestru. Następnie czyścimy 6 młodszych bitów rejestru - są to maski wyjątków. Na koniec wczytujemy zawartość rejestru `rax` do Control Word.

Funkcja `maskujWyjatki` modyfikuje ustawienia maskowania wyjątków w Control Word. Typ wyjątku jest zapisany w rejestrze `rdi`. Za pomocą instrukcji logicznej `or` ustawiamy odpowiednie maski wyjatków.

Na końcu znajduje się funkcja mająca za zadanie wywołać wyjątek dzielenia przez zero. Instrukcją `fldz` zostaje załadowane 0 do stosu FPU. Następnie do stosu ładujemy 1. Za pomocą instrukcji `fdivp` dzielimy ST(0)/ST(1). Rozkaz `fstp` usuwa ostatnią wartość ze stosu.

```
.bss
.comm controlWord, 2

.text
#deklaracje funkcji

sprawdzWyjatki:
    mov $0, %rax
    fstsw %ax
    fwait
    and $0x0ff, %ax
ret

wyczyscMaski:
    fstcw controlWord
    fwait
    mov controlWord, %ax
    and $0xffc0, %ax
    mov %ax, controlWord
    fldcw controlWord
ret

maskujWyjatki:
    fstcw controlWord
    fwait
    mov controlWord, %ax
    or %di, %ax
    mov %ax, controlWord
    fldcw controlWord
ret

dzieleniePrzezZero:
    fldz
    fld1
    fdivp
    fstp %st
ret
```

Kod C:

Jeśli chcemy sprawdzić wyjątki to za pomocą przesunięć bitowych w prawo dostajemy się do każdego bitu i operacją modulo 2 sprawdzamy czy jest ustawiony.

Następnie możemy zamaskować jakiś wyjątek poprzez wybranie odpowiedniej opcji w menu. Funkcja `maskujWyjatki` przyjmuje argument typu int - czyli cyfrę 2 podniesionej do potęgi równej wybranej masce. Dzięki temu działaniu otrzymujemy interesujący nas bit.

Możemy również wyczyścić wszystkie maski albo wywołać wyjątek.

```
// biblioteki
// deklaracje funkcji
int main(void){
int choice, exceptions;
do{
    // wyswietlenie menu
    scanf("%d", &choice);

    if (choice == 1){
        exceptions = sprawdzWyjatki();
        if (exceptions == 0){
            printf("Brak wyjatkow\n");
            continue;
        }
        if (exceptions % 2 == 1)
            printf("Invalid-Operation Exc\n");
        exceptions = exceptions >> 1;
        if (exceptions % 2 == 1)
            printf("Denormalized-Operand Exc\n");
        exceptions = exceptions >> 1;
        if (exceptions % 2 == 1)
            printf("Zero-Divide Exc\n");
            exceptions = exceptions >> 1;
        if (exceptions % 2 == 1)
            printf("Overflow Exc\n");
        exceptions = exceptions >> 1;
        if (exceptions % 2 == 1)
            printf("Underflow Exc\n");
        exceptions = exceptions >> 1;
        if (exceptions % 2 == 1)
            printf("Precision Exc\n");
    }
    else if (choice == 2){
    // wybor wyjatku do zamaskowania
        scanf("%d", &exceptions);
        if (exceptions > 5||exceptions<0){
            printf("Niepoprawny wyb r\n");
            continue;
        }
         maskujWyjatki((int)pow(2,exceptions));
    }
    else if (choice == 3){
        wyczyscMaski();
    }
    else if (choice == 4){
        dzieleniePrzezZero();
    }
} while(choice != 0);
}
```

6.2.2 Szereg Taylora

Kod ASM:

Za pomocą szeregu Taylora możemy z odpowiednią dokładnością przybliżyć wartość funkcji. Im więcej kroków tym dokładniejszy wynik. 1 to wartość pierwszej silni. Funkcja przyjmuje kolejne argumenty: w `rdi` liczbę wyrazów ciągu, a w `xmm0` niewiadomą (x, czyli wykładnik potęgi).
Kolejne rejestry jednostki FPU:
ST(2) = niewiadoma, x
ST(1) = aktualna suma ciągu (początkowo równa x)
ST(0) = aktualny wyraz ciągu (początkowo równy x)

W pętli za pomocą instrukcji `fmul` mnożymy poprzedni wyraz ciągu przez x, wynik w ST(0) to aktualny wyraz ciągu.
Następnie przechodzimy do mianownika. Kolejne rejestry jednostki FPU:
ST(2) = 1
ST(1) = wynik silni (początkowo 1)
ST(0) = numer wyrazu silni
Następnie zwiększamy poprzedni numer silni o jeden i mnożymy powstały wynik przez wynik silni.
Za pomocą instrukcji `fstpl` zostaje zapisany ostatni wyraz silni do zmiennej i zostaje ściągnięty ze stosu FPU. Następnie za pomocą `fxch` i `fstp` sprzątamy nasz stos.
Aktualny stan rejestrów:
ST(3) - niewiadoma, x
ST(2) - aktualna suma ciągu
ST(1) - mianownik (aktualny wyraz ciągu)
ST(0) - silnia (aktualny dzielnik)

Następnie za pomocą `fdivr` dzielimy obecny wyraz przez silnię. Później usuwamy ze stosu obecny dzielnik, dodajemy wartość obecnego wyrazu ciągu do wyniku ogólnego.

Na koniec przenosimy przez stos naszą sumę ciągu, która znajduje się w rejestrze ST(0) do rejestru xmm0.

```
.data
silnia: .double 1.0

.text
# deklaracja funkcji

taylor:
    push %rbp
    mov %rsp, %rbp
    sub $8, %rsp
    movsd %xmm0, (%rsp)
    fldl (%rsp)
    fld %st
    fld %st

    movq $0, %rsi
    fwait

    petla:
        cmp %rdi, %rsi
        je koniec
        inc %rsi

        fmul %st(2), %st

        fld1
        fld1
        fldl silnia

        fadd %st(2), %st
        fmul %st, %st(1)
        fstpl silnia

        fxch %st(1)
        fstp %st

        fdivr %st, %st(1)
        fstp %st
        fadd %st, %st(1)
        jmp petla
    koniec:
    fstp %st
    fstpl (%rsp)
    movsd (%rsp), %xmm0

mov %rbp, %rsp
pop %rbp
ret
```

Kod C:

```
#include <stdio.h>
extern double taylor(double niewiadoma, int iteracje);

int main(void)
{
double niewiadoma;
int iteracje;

printf("Niewiadoma, czyli pot ga (e^x): ");
scanf("%lf", &niewiadoma);
printf("Liczba wyrazow ciagu: ");
scanf("%d", &iteracje);
printf("Przybli any wynik to: %lf\n", taylor(niewiadoma, iteracje)+1);
return 0;
}
```

6.3 Podsumowanie i wnioski

Wszystkie programy zostały uruchomione i działały prawidłowo. W przypadku korzystania ze stosu FPU warto sobie zapisywać stany rejestrów w konkretnych chwilach.

7 Bibliografia

1. „Professional Assembly Language” Richard Blum
2. „Programming from the Ground Up” Jonathan Bartlett
3. en.wikibooks.org
4. http://www.zak.ict.pwr.wroc.pl/materials/architektura/
5. https://ww2.ii.uj.edu.pl/ kapela/pn/print-lecture-and-sources.php