Wrocław, dn. 20 kwietnia 2017

Hanna Grodzicka, 226154

Łączenie różnych języków programowania w jednym projekcie

sprawozdanie z laboratorium przedmiotu „Architektura Komputerów 2”

Rok akad. 2016/2017, kierunek: INF

Prowadzący:

mgr inż. Aleksandra Postawka

1. Cel ćwiczenia

Ćwiczenie miało na celu naukę łączenia języka Asemblera z językiem C.

Pierwsze ćwiczenie to program w języku Asemblera, w którym powinna zostać wywołana funkcja napisana w języku C, znajdująca się w innym pliku. Funkcja w C miała przyjmować trzy argument - typu *integer*, *float* oraz *bool*. W zależności od ostatniej zmiennej, funkcja miała wyświetlić proste działanie arytmetyczne z użyciem dwóch pierwszych parametrów. W asemblerowym programie należało skorzystać z funkcji bibliotecznych *printf* i *scanf* do wyświetlania i pobierania argumentów zmienno- i stałoprzecinkowych.

Drugi program powinien był zostać napisany w języku C. Tym razem to funkcja w języku Asemblera (z osobnego pliku) miała być wywoływana. Funkcja dokonuje konwersji z tekstowej reprezentacji szesnastkowej na liczbę (dziesiętną). Jako argumenty powinna przyjmować wskaźnik na ciąg znaków, a także można było przekazywać długość tego ciągu.

Trzecie zadanie to program w języku C ze wstawką w języku Asemblera. Wymogiem było użycie stałej stałoprzecinkowej np. jako wskaźnika na łańcuch znaków. Program powinien dokonywać konwersji ciągu cyfr na podstawie przyjętego klucza, który zostaje dodany do cyfry. Przechodzenie cyfr po dodaniu klucza jest analogiczne jak przy szyfru Cezara.

2. Przebieg ćwiczenia

Wszystkie programy były kompilowane za pomocą `gcc` - kompilatora C w ramach projektu GNU - z użyciem opcji -g (dostarcza informacji dla `gdb` przy debuggowaniu) oraz -o (do utworzenia pliku wynikowego).

A. Program w języku Asemblera

Funkcja w języku C (w pliku o nazwie *fun.c*):

```
#include <stdio.h>
#include <stdbool.h>    //konieczne do użycia boola

int fun(int x, float y, bool z) {
    if(z)
        return x*x*x + y*y*y;
    else
        return x*x + y*y;
}
```

Biblioteka *stdio.h* jest dołączana, aby możliwe było skorzystanie z wywołania funkcji *scanf* i *printf* w pliku z programem.

Program w języku Asemblera:

W sekcji `.data` deklarowane są stałe kolejno: do wczytywania lub wypisywania wartości dziesiętnych, do wczytywania lub wypisywania wartości zmiennoprzecinkowych oraz do wyświetlenia nowej linii. Dyrektywa `.asciz` jest tożsama z `.ascii`, z tym wyjątkiem, że na koniec podanego stringa dodawany jest zero (nul character, \0)- litera „z” w „`.asciz`” jest od słowa „zero”.

```
.data
decimal: .asciz "%d"
float:   .asciz "%f"
nl:      .asciz "\n"
```

Poniższe bufory będą służyły o przechowywania podawanych przez użytkownika argumentów, które później zostaną podane do wywołania funkcji w języku C.

```
.bss
.comm number1, 8    #int
.comm number2, 8    #float
.comm bool, 1       #bool

.text
.global main
main:
```

Do wywołania funkcji *scanf* z biblioteki *stdio.h* należy podać argumenty do odpowiednich rejestrów:

- w `rax` znajduje się liczba przekazywanych argumentów zmiennoprzecinkowych;
- w `rdi` format liczby, który określa się tak jak przy zwykłej funkcji *scanf* np. %s;
- w `rsi` wskaźnik na miejsce w pamięci, w które zostanie wpisany pobrany znak.

```
## WCZYTYWANIE LICZB ##
# scanf("%d", &number1);
movq    $0, %rax          #0 parametrów zmiennoprzecinkowych
movq    $decimal, %rdi    #format zapisania liczby w buforze
movq    $number1, %rsi    #adres bufora do zapisania wyniku
call    scanf             #<stdio.h>
```

Do wywołania funkcji *printf* należy podać:

- w `rax` liczbę przekazywanych argumentów zmiennoprzecinkowych;
- w `rdi` format liczby (zmiennej) do wypisania;
- w `rsi` adres w pamięci, z którego znak zostanie wypisany na wyjście standardowe.

```
# printf(number1);
movq    $0, %rax          #0 parametrów zmiennoprzecinkowych
movq    $decimal, %rdi
movq    number1, %rsi
call    printf
```

Wyświetlenie znaku nowej linii, który nie wymaga podawania formatu zmiennej (bo jej nie ma):

```
# printf("\n");
movq    $0, %rax
movq    $nl, %rdi
call    printf
```

Pobranie liczby zmiennoprzecinkowej przyjmuje wartość 0 do `rax`, ponieważ nie przekazujemy żadnej takiej liczby przed wywołaniem funkcji.

```
# scanf("%f", &number2);
movq    $0, %rax
movq    $float, %rdi
movq    $number2, %rsi
call    scanf
```

Pobranie zmiennej typu *bool*:

```
# scanf("%d", &bool);
movq    $0, %rax          #0 parametrów zmiennoprzecinkowych
movq    $decimal, %rdi    #format zapisania liczby w buforze
movq    $bool, %rsi       #adres bufora do zapisania wyniku
call    scanf             #<stdio.h>


## WYWOŁANIE FUNKCJI W C ##
```

Do wywołania wcześniej napisanej funkcji potrzebne są trzy argumenty. Pierwszy z nich, typu *integer*, jest zapisany w buforze `number1`. Jego zawartość należy umieścić w odpowiedni rejestr, którym jest w tym przypadku `rdi`, czyli rejestr przyjmujący pierwszy argument. Rejestrem na drugi argument jest rsi. Do niego przekazana zostaje zawartość z bufora bool. Drugi argument funkcji (liczba typu *float*) nie trafia do `rsi` ze względu na swój format. Przy pomocy polecenia `movss` możliwe jest przekopiowanie tej liczby do specjalnego rejestru o nazwie `xmm0`. Takich rejestrów jest łącznie osiem z numeracją do `xmm7` i każdy z nich może pomieścić 128 bitów. Służą one do przechowywania liczb zmiennoprzecinkowych i

informacja w `rax` o liczbie przekazywanych parametrów zmiennoprzecinkowych jest konieczna, aby odpowiednia liczba tych rejestrów była brana pod uwagę przy wywołaniu funkcji.

```
# argumenty
movq    $0, %rdi #czyszczenie rejestru przed włożeniem liczby
movq    $0, %rbx                #licznik number1
movq    number1(,%rbx,8), %rdi  #pobranie liczby int z bufora
movq    bool, %rsi
movss   number2, %xmm0       #single-precision floating-point

# wywołanie funkcji fun z fun.c
movq    $1, %rax       #1 parametr zmiennoprzecinkowy w %xmm0
call    fun
```

Konwersja *float*a na typ *double* jest potrzebna, aby skorzystać z funkcji *printf*.

```
cvtps2pd %xmm0, %xmm0   #konwersja wyniku na double-precision
                        #floating-point

## WYPISANIE WYNIKU ##
```

Przy podawaniu parametrów zmiennoprzecinkowych przed wywołaniem funkcji *printf* konieczne jest zrobienie odpowiedniej rezerwy miejsca na stosie, która jest zależna od liczby tych parametrów. Są one odkładane na stos, a w czasie wywołania funkcji istnieje ryzyko ich nadpisania. Dla dwóch parametrów taka rezerwa wynosiłaby 16 bajtów.

```
# printf(wynik);
movq    $1, %rax      #1 parametr - %xmm0
movq    $float, %rdi
subq    $8, %rsp      #rezerwa, żeby printf nie nadpisał
                      #liczby zmiennoprzecinkowej na szczycie
call    printf
```

```
addq    $8, %rsp      #powrót do poprzedniego stanu

# printf("\n");
movq    $0, %rax
movq    $nl, %rdi
call    printf

## EXIT ##
movq    $0, %rax
call    exit
```

B. **Program w C**

Funkcja w języku Asemblera:

Jeśli funkcja ma coś zwracać, to dla liczb stałoprzecinkowych użyty będzie rejestr `rax`, a dla zmiennoprzecinkowych `xmm0`.

```
.data
.text
```

Zdefiniowanie funkcji *fun*:

```
.globl fun
.type fun, @function
```

Argumenty funkcji są przekazywane przez te same rejestry (analogia z umieszczaniem argumentów w odpowiednich rejestrach przed wywołaniem funkcji). W systemie 64-bitowym dla pierwszego argumentu jest to `rdi`, dla drugiego `rsi`, dla trzeciego `rdx` itd.

```
# fun   (char*,   int)
# %rax   %rdi,    %rsi

fun:
pushq   %rbp            #standardowe operacje
movq    %rsp, %rbp
```

Część logiczna (konwersja z systemu szesnastkowego na dziesiętny) jest zbudowana częściowo o kod z programu z szyfrem Cezara.

```
movq    $0, %r9         #licznik
movq    $0, %rax
movq    $0, %rbx        #do pobrania znaku
```

W pętli odbywa się:

- pobranie znaku w ascii;
- przekodowanie go na wartość liczbową od 0 do 15;
- dodanie wartości liczbowej do rejestru z sumą całej liczby;
- przemnożenie całej sumy przez podstawę systemu, z którego jest konwertowana, czyli przez 16.

Konwencja każdorazowego przemnażania przez 16 jest oparta o schemat Hornera.

```
loop:
imul    $16, %rax
movb    (%rdi, %r9, 1), %bl
```

Jeśli napotkany znak nie będzie żadnym ze znaków służących do zapisu liczby szesnastkowej, to funkcja kończy swoje działanie.

```
cmpb    $'A', %bl
jl      czy_to_cyfra
cmpb    $'Z', %bl
jg      return
#kod od tego miejsca się wykona <=> duża litera w %al
subb    $55, %bl
jmp     wypisz

czy_to_cyfra:
cmpb    $'0', %bl
jl      return
cmpb    $'9', %bl
jg      return
subb    $'0', %l
```

Suma jest zapisywana od razu do rejestru `rax`, bo w nim będzie zwracana.

```
wypisz:
addq    %rbx, %rax
incq    %r9         #przejście do następnego znaku z bufora
cmpq    %rsi, %r9   #porównanie licznika z drugim argumentem
jl      loop

return:
movq    %rbp, %rsp  #standardowe operacje
popq    %rbp
ret
```

Program w języku C:

```
#include <stdio.h>       //do printf i scanf
#include <string.h>      //do strlen()
```

Funkcję zdefiniowaną poza plikiem z programem określa się przy pomocy „extern”.

```
extern int fun(char*, int); //funkcja w jęz. Asemblera

char txt;
int txt_len, output;

int main() {
    printf("Wpisz tekst: ");
    scanf("%s", &txt);
    txt_len = strlen(&txt);

    output = fun(&txt, txt_len);
    printf("%d\n", output);

    return 0;
}
```

Program pobiera od użytkownika ciąg znaków, oblicza jego długość, następnie wywołuje funkcję napisaną w języku Asemblera podając na argumenty adres początku ciągu znaków oraz jego długość. Zwracana wartość liczbowa w `rax` jest przepisywana do zmiennej typu *integer* o nazwie `output`, a później wyświetlana.

C. Program w języku C ze wstawką w języku Asemblera

```
#include <stdio.h>
#include <string.h>

static char txt;        //stała
int txt_len, key;

int main(void)
{
    printf("Wpisz tekst: ");
    scanf("%s", &txt);
    txt_len = strlen(&txt);
    printf("Wpisz klucz: ");
    scanf("%d", &key);
```

Wstawkę w języku asemblera można zapisać na dwa sposoby:

- `asm(„#działania”);`
- `__asm__(„#działania”);`

W samej wstawce każdą linijkę kodu z Asemblera trzeba zapisać w cudzysłowach, a przed zakończeniem cudzysłowiu pisać znak nowej linii tj. „\n” lub średnik. Oprócz tego nazwy wszelkich rejestrów muszą być poprzedzone dodatkowym znakiem procentu.

```
    //wstawka w języku Asemblera
    asm(
        "movq   $0, %%rbx \n"       //licznik
        "movq   $0, %%rax \n"       //bufor na znak
        "loop: \n"
```

```
"movb   (%0, %%rbx, 1), %%al \n"    //pobranie znaku
"cmpb   $'0', %%al \n"
"jl     next \n"
"cmpb   $'9', %%al \n"
"jg     next \n"
"subb   $'0', %%al \n"              //ascii -> int
"addl   %2, %%eax \n"               //int + klucz
"movq   $0, %%rdx \n"
"movq   $10, %%rcx \n"
"idiv   %%rcx, %%rax\n"
"addq   $'0', %%rdx \n"
"movb   %%dl, (%0, %%rbx, 1) \n"
"next: \n"
"incq   %%rbx\n"
"cmpl   %1, %%ebx\n"
"jl     loop \n"
```

W asemblerowej wstawce z `txt` pobierany jest znak. Jest on przekodowywany na wartość liczbową, następnie zostaje dodany do niego klucz i całość jest dzielona przez 10. Wynik z dzielenia (w `rax`) jest przekodowywany z powrotem na ascii i umieszczany pod tym samym adresem w buforze.
Poniżej, jest część, która jest opcjonalna w asemblerowej wstawce - można ją pisać, choć nie zawsze jest to konieczne.
Po każdym z trzech dwukropków należy napisać kolejno:
- zmienne, w których zwracany jest wynik działania (wyjściowe);
- zmienne, do których odnosimy się we wstawce (wejściowe);
- rejestry używane we wstawce.

```
:
:"r"(&txt), "r"(txt_len), "r"(key)  //%0, %1, %2
:"%rax", "%rbx", "%rcx", "%rdx"

);
```

Zmienne wejściowe i wyjściowe zostają umieszczone w rejestrach i można się do nich odnosić za pomocą znaku procenta z odpowiednią cyfrą np. %0, %1. Sama cyfra zależy od kolejności deklaracji zmiennych po dwukropku. Indeksuje się je od zera począwszy od pierwszego dwukropka, czytając od lewej do prawej i kończąc na drugim dwukropku.
W cudzysłowach, przed samymi zmiennymi, które są przekazywane jak w funkcji, można narzucić rejestr, do którego zmienna ma być pobrana.

r	General Purpose Register (dowolny dostępny)
a	rax
b	rbx
c	rcx
d	rdx
S	rsi
D	rdi

Przy deklaracji zmiennych wyjściowych powinno się jeszcze dopisać znak równości np. `"=r"(val)`.
Wypisanie wyniku i zakończenie programu:

```
    printf("Wynik: %s\n", &txt);
    return 0;
}
```

3. Podsumowanie i wnioski

Łączenie kodu Asemblera z językiem C nie jest wyłącznie ćwiczeniem dydaktycznym. Skomplikowane algorytmy są często pisane w Asemblerze, ponieważ zapewniają dużą wydajność. Ponadto języki wysokiego poziomu rzadko mają stworzony dostęp do wywoływania funkcji systemowych, dlatego używany jest Asembler w takich momentach.

4. Bibliografia

- https://pl.wikipedia.org/wiki/Asembler_x86
- https://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C
- https://en.wikipedia.org/wiki/Scanf_format_string
- https://en.wikipedia.org/wiki/Inline_assembler