Marta Głowacka, 234999 Wrocław, 7.05.2018r.

WT-N-10

LABOLATORIUM ARCHITEKTURY KOMPUTERÓW

4. Łączenie kodu języka C z asemblerem

1. Treść ćwiczenia

Zakres i program ćwiczenia:

- Dostęp do języka C z poziomu kodu asemblera
- Wykorzystanie wstawki asemblerowej w kodzie w języku C

Zrealizowane zadania:

- Program w języku asemblera: wczytanie liczb typu integer, double i float za pomocą pojedynczego wywołania funkcji scanf
- Wywołanie funkcji dodającej w języku C double(int x, float y, double z)
- Wypisanie wyniku za pomocą funkcji bibliotecznej printf
- Wstawka asemblerowa konwertująca liczbę do łańcuch znaków w reprezentacji ósemkowej (zmienne globalne w C)
- Wywołanie funkcji napisanej w języku asm, która znajduje wszystkie czynniki pierwsze danej liczby i zapisuje je pod danych adresem i zwraca ich liczbę

2. Przebieg ćwiczenia

Program w języku asemblera

Będziemy najpierw chcieli wczytać liczby różnych typów pojedynczym wywołaniem funkcji C z biblioteki studio.h (instrukcja scanf).

Aby wczytać liczby różnych typów (int, float oraz double), należy sprecyzować ich formaty. Dla int będzie to %d, dla float %f, a dla double %d. Dla przykładu pokazano jak wygląda format dla każdego typu, ale w programie użyto jedynie format_x oraz format_l (do odczytu wyniku, który jest typu double).

```
format_d: .asciz "%d"
format_f: .asciz "%f"
```

```
format_l: .asciz "%lf"
format_x: .asciz "%d%f%lf/n"
nowa_linia: .asciz "/n"
```

Tworzymy bufory na liczby, podając ich wielkość zależnie od typu.

```
.bss
.comm liczba1, 4        #Bufory na liczby typu int, float i
double
.comm liczba2, 4
.comm liczba3, 8
```

Po etykiecie _start lub main zaczynamy wczytywanie. Kolejne argumenty umieszczamy kolejno w rejestrach rax (liczba argumentów zmiennoprzecinkowych – przesyłamy zero), rdi (format w jakim ma być zapisany wynik), rsi, rdx i rcx (kolejne liczby). Na koniec wywołujemy funkcję systemową scanf poprzez instrukcję call.

```
movq $0, %rax
movq $format_x, %rdi
movq $liczba1, %rsi
movq $liczba2, %rdx
movq $liczba3, %rcx
call scanf
```

Kolejnym krokiem jest wywołanie napisanej przez nas funkcji w języku C. Wygląda ona tak:

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

Wywołujemy funkcję z parametrami stałoprzecinkowym oraz zmiennoprzecinkowymi.

```
movq $1, %rax            #Jeden arg zmiennoprzecinkowy -
przesyłany jeden parametr w rejestrze XMM0
movq $0, %rdi            #Czyszczenie rdi
movq $0, %rcx            #Licznik, by adresować pamieć poniżej
mov liczba1(, %rcx, 4), %edi    #Przeniesienie pierwszego
parametru - typ int - do rdi
```

Argumenty zmiennoprzecinkowe umieszczamy w specjalnych rejestrach xmm0 oraz xmm1.

```
movss liczba2, %xmm0        #Przeniesienie drugiego parametru
- typu zmiennoprzecinkowego do rejestru XMM0
movsd liczba3, %xmm1        #Przeniesienie drugiego parametru
- typu zmiennoprzecinkowego do rejestru XMM0
```

Następnie instrukcją call wywołujemy funckję dodawania.

```
dod:
call dodawanie
```

Ostatnie, co należy zrobić, to wyświetlić wynik instrukcją printf. W tym celu znów przekazujemy odpowiednie argumenty (w raz liczba argumentów zmiennoprzecinkowych – 1 – sam wynik typu double, a jego format to format_l zdefiniowany na początku).

```
mov $1, %rax            #Jednen parametr zmiennoprzecinkowy -
liczba do wyswietlenia w rejestrze XMM0
mov $format_l, %rdi
sub $8, %rsp            #Żeby printf nie zmienił wartości
ostatniej komórki na stosie
call printf
add $8, %rsp
```

Musimy pamiętać o tym, aby printf nie zmienił ostatniej komórki na stosie – dlatego wskaźnik na stos przesuwamy o 8 (tyle zajmuje typ double).

```
#===Znak nowej linii===
movq $0, %rax            #Nie przesyłamy parametrów
zmiennoprzecinkowych
movq $nowa_linia, %rdi        #Parametr typu int
                #wsk na ciąg znaków do wyświetlenia
                #Znak nowej linii
call printf
```

Na koniec dodaliśmy znak nowej linii i zamykamy program.

```
#===Koniec programu===
mov $SYSEXIT, %rax
mov $EXIT_SUCCESS, %rdi
syscall
```

Program w języku C ze wstawką asemblerową

Należy zadeklarować dwa bufory – jeden posłuży do przechowywania odwróconej liczby (gdyż musimy uzyskiwać cyfry od końca – od jedności), a drugi do jej postaci zapisanej w poprawnej kolejności.

```c
#include <stdio.h>

// Deklaracja zmiennej przechowującej ciąg znaków do konwersji
int liczba = 1233;
char buf[10] = {0};
char textinv[10] = {0};

int main(void)
{
```

```
    //
    // Wstawka Asemblerowa
    //
    asm(
```

W eax znajduje się wprowadzona liczba, w rbx znajduje się podstawa systemu, czyli 8.

```
"movq $0, %%rax \n"
"movl %0, %%eax \n"         //zapisanie wyniku globalnego do raxa
"movq $8, %%rbx \n"         //zapisanej podstawy wyjściowej do rbx (rbx zawsze trzyma
podstawy)
"movq $0, %%rcx \n"         //wyzerowanie rcx - licznik
"jmp zamiana_na_system_wyj \n"
```

Należy zamienić każdą cyfrę na jej reprezentację w systemie ósemkowym, robimy to poprzez wielokrotne dzielenie przez 8 w pętli.

```
"zamiana_na_system_wyj: \n"
"movq $0, %%rdx\n"          //rdx wcześniej trzymał B/2 - 1
"div %%rbx\n"               //dzielenie bez znaku liczby z rejestru rax (wynik
globalny) przez rbx
                            //(podstawa wyjściowego systemu), zapis wyniku do rax, a
reszta z dzielenia
                            //do rdx
```

Kodujemy cyfrę na ASCII.

```
//Reszta zapisana w rdx to część wyniku po konwersji
"add $0x30, %%rdx\n"        //Dodanie kodu znaku pierwszej liczby - zakodowanie w ascii
"mov %%dl, (%2,%%rcx,1)\n" //zapisanie z dl do bufora w odwrotnej kolejności

"inc %%rcx\n"                           //zwiększenie licznika
"cmp $0, %%rax\n"                       //czy wynik dzielenia jest juz zerowy
"jne zamiana_na_system_wyj\n"                   //jeśli nie, rób pętle zamiany na
system dalej
"jmp odwrocenie_kolejnosci_przygotowanie\n"     //jeśli tak, odwróć kolejność
```

Później musimy odwrócić kolejność cyfr w buforze.

```
"odwrocenie_kolejnosci_przygotowanie:\n"

"movq $0, %%rdi\n"
"movq %%rcx, %%rsi\n"             //zapisanie do licznika rsi tego, co bylo w liczniku
rcx
                            //jest tam liczba cyfr zamienionej liczby na system
wyjsciowy
"dec %%rsi\n"                       //odejmujemy ostatni przypadek, w którym wynik
okazuje sie juz zerowy
                            //tego zera nie należy rozpatrywać
"jmp odwrocenie_kolejnosci\n"

"odwrocenie_kolejnosci:\n"
"movq (%2,%%rsi,1), %%rax\n"            //zapisznie do raxa ostatniej cyfry wyniku
"movq %%rax, (%1, %%rdi, 1) \n"   //z raxa do wynikowego bufora
                                  // w ten sposób tworzona jest dobra kolejność

"inc %%rdi\n"               //zwiększenie licznika pętli
```

```
"dec %%rsi\n"              //zmniejszenie licznika, który liczy cyfry
"cmp %%rcx, %%rdi\n"       //porównanie licznika pętli z wartością ile razy pętla ma
się wykonać, a ma się
                  //wykonać tyle razy, ile jest cyfr w systemie wyjściowym - liczba
tych cyfr w rcx

"jle odwrocenie_kolejnosci\n"
```

Na koniec wstawki należy wstawić używane rejestry (parametry wej i wyj).

```
    :// Nie mamy żadnych parametrów wyjściowych.

    :"r"(liczba), "r"(&buf), "r"(&textinv) // Lista parametrów wejściowych - zmiennych
które zostaną
    // zapisane do rejestrów i będzie możliwy ich odczyt w kodzie asm

    :"%rax", "%rbx", "%rcx", "%rdx", "%rsi", "%rdi" // Rejestry których będziemy
używać w kodzie asemblerowym.
    );
```

Wypisanie wyniku i zakończenie.

```
    //Wyświetlenie wyniku
    printf("Wynik: %s\n", buf);

    return 0;
}
```

Wnioski

Nie należy przesyłać liczby przez referencję oraz należy wcześniej wyzerować rejestr rax. Do zapisu liczby należy użyć rejestru eax, gdyż liczba integer składa się z 4 bajtów (32 bity).

Aby używać rejestrów we wstawce asemblerowej, należy postawić przed nim `%%`.

Źródła:

Professional Assembly Language, Richard Blum