Marta Głowacka, 234999 Wrocław, 23.04.2018r.

WT-N-10

LABOLATORIUM ARCHITEKTURY KOMPUTERÓW

2. Funkcje rekurencyjne, stos

1. Treść ćwiczenia

Zakres i program ćwiczenia:

- Użycie funkcji oraz rekurencji
- Posługiwanie się stosem

Zrealizowane zadania:

- Napisanie funkcji zwracającej indeks początku najdłuższego ciągu zer w danym łańcuchu znaków
- Funkcja rekurencyjna, w której wynik i argument będzie przekazywany przez 1) rejestry 2) stos.

$$\begin{cases} x_0 = 3 \\ x_n = -5x_{n-1} + 7 \end{cases}$$

2. Przebieg ćwiczenia

Funkcja – łańcuch zer

Wstęp

Aby zrozumieć funkcję napisaną w asemblerze, najlepiej prześledzić jej działanie na przykładzie kodu w C++ - nie jest to napisana funkcja, przyjmująca ciąg znaków *char jako argument, gdyż w tym przykładzie zależy nam szczególnie na zrozumieniu działania pętli i wiedzy, jakiej długości jest tablica, i jakimi wartościami dokładnie jest wypełniona. Kod ten nie odwzorowuje dokładnie kodu napisanego w assemblerze, ponieważ specyfikacja tych języków jest inna (np. w assemblerze długość ciągu zczytujemy z rejestru rax - movq $SYSREAD, %rax, a w C++ rozmiar tablicy podajemy w przykładzie jako const).

```
#include <iostream>

using namespace std;

int main()
{
	const int dlugoscCiagu = 14;
	char tab[dlugoscCiagu] = { 'x', '0', '0', 'x', 'x', '0', '0', '0', '0', '0',
'v', '0', '0', '0' };
	int liczbaZer = 0, poprzedniaLiczbaZer = 0;

	int index = -1;

	for (int i = 0; i < dlugoscCiagu; i++)
	{
		if (tab[i] == '0')
		{
			while (tab[i] == '0')
			{
				liczbaZer++;
				i++;
			}
			if (poprzedniaLiczbaZer < liczbaZer) {
				poprzedniaLiczbaZer = liczbaZer; index = i - liczbaZer;
			}
			liczbaZer = 0;
		}
	}

	cout << "najwiekszy ciag zer: " << poprzedniaLiczbaZer;
	cout << "indeks gdzie sie zaczyna najdluzszy ciag zer: " << index;
	getchar();
	return 0;

}
```

Krótki komentarz kodu: jeśli w ciągu nie będzie żadnego zera, wypiszemy w indeksie wartość -1. Na kod składają się dwie pętle – jedna zagnieżdżona w drugiej. Pętla for przechodzi kolejne wartości w podanym ciągu ASCII. Pętla while przechodzi ciągi samych zer. Zaczyna to robić dopiero wtedy, gdy natrafimy na pierwsze zero takiego wewnętrznego ciągu. Po pętli for zapisywane są niezbędne informacje: sprawdzamy, czy wcześniejszy maksymalnie długi ciąg zer nie jest mniejszy od właśnie sprawdzonego. Jeśli tak, maksymalnym ciągiem zer będzie ten właśnie sprawdzany. Zapisujemy indeks rozpoczęcia takiego najdłuższego ciągu zer.

W sekcji **.data** inicjujemy funkcje systemowe. Zapisujemy kod ASCII zera, długości buforów, komentarze do wypisania w konsoli itp.

```
seqSize = 64

SEARCHED = 48                    #Kod ascii zera
```

```
komunikat_dl: .ascii "Dlugosc najdluzszego ciagu zer: "
komunikat_dl_len = .-komunikat_dl

komunikat_ind: .ascii "Znajduje sie pod indeksem: "
komunikat_ind_len = .-komunikat_ind
```

W sekcji **.bss** inicjujemy bufory, które posłużą nam do wczytania sekwencji podanej przez użytkownika. Tworzymy również bufory, które posłużą nam do wypisania wyniku (indeksu najdłuższego ciągu zer oraz jego długości – dodatkowo).

```
.bss
.comm seqWithZeros, seqSize
.comm out, 32
.comm out_ind, 32

.comm out_inv, 32
.comm out_ind_inv, 32
```

Po etykiecie **_start** przechodzimy do wczytania ciągu używając funkcji systemowej - SYSREAD. Długość ciągu zapiszemy w rejestrze %r8. Odejmiemy od niego dwa, gdyż nie bierzemy pod uwagę znaku '/n' i chcemy liczyć od 0, jak w tablicy.

```
#===Dlugosc ciagu w rejestrze r8===
movq %rax, %r8
sub $2, %r8                    #liczymy od zera
call funkcja_zer
```

Na koniec wywołujemy funkcje, która zlicza zera. Umieszczona jest ona na końcu programu, ale dla zachowania logiki przejść, przytoczymy ją teraz.

Funkcja rozpoczyna się etykietą i kończy ret – dzięki tej instrukcji powrócimy do miejsca w programie, z którego funkcja została wywołana.

Oznaczmy, sobie który rejestr za co odpowiada.

rsi - i – licznik w pętli

r9 - licznik liczby zer w aktualnym ciągu zer

r10 - liczba poprzednich zer (najdłuższego, poprzedniego ciągu zer)

r11 - indeks pierwszego zera najdluzszego ciagu

```
funkcja_zer:
movq $0, %rsi
movq $(-1), %r11
movq $0, %r10
movq $0, %r9
```

Dokonaliśmy inicjalizacji naszych zmiennych.

Rozpoczynamy pętlę sprawdzającą – odpowiednik pętli for w C++.

Pętla wykonuje sie aż do osiągnięcia końca całego ciągu.

W środku pętli znajduje się kolejna – zagnieżdżona – odpowiednik pętli while w kodzie C++.

Pętla ta działa podobnie jak pętla już wcześniej omówiona, jedynie reprezentacja instrukcji nieco się różni np. zamiast użycia tab[i], aby dostać się do kolejnego elementu tablicy ASCII, używamy bufora seqWithZeros, po którym przesuwamy się licznikiem rsi, a wynik wpisujemy do 8 bitowego rejestru al.

```
petla_sprawdzajaca:

    cmp %r8, %rsi
    jg zmiana_na_ascii

    movb seqWithZeros(, %rsi, 1), %al
    inc %rsi

    cmp $SEARCHED, %al
    jne petla_sprawdzajaca

    #===Petla zagniezdzona===
    petla_zliczajaca_zera:
```

```
          cmp $SEARCHED, %al
          jne przypisanie
          inc %r9

          movb seqWithZeros(, %rsi, 1), %al
          inc %rsi

     jmp petla_zliczajaca_zera          #Jesli     licznik     <
dlugosci

przypisanie:
     cmp %r10, %r9
     jl zerowanie

     movq %r9, %r10
     movq %rsi, %r11
     dec %r11
     sub %r9, %r11
zerowanie:
     movq $0, %r9

jmp petla_sprawdzajaca
ret
```

Po tej operacji, wynik, czyli indeks pozycji najdłuższego ciągu zer znajduje się w rejestrze r11, a liczba zer w tym ciągu w r10. Teraz należy jedynie zamienić wartości liczbowe z tych rejestrów na ASCII. Pomogą nam w tym bufory out i out_ind. Musimy wykonać, powtarzane już wcześniej, operacje dzielenia przez 10 (aby uzyskać kolejne cyfry liczby), a następnie zamiany każdej cyfry na ascii poprzez dodanie wartości 48.

Kod tych operacji był omawiany już wiele razy w poprzednich sprawozdaniach i jest dołączony w załączniku z kodem.

Na koniec do buforów musimy dodać znak końca linii:

```
koniec_linii:
movb $0x0A, out(, %rcx, 1)
movb $0x0A, out_ind(, %r12, 1)
inc %rcx
inc %r12
```

A później wypisujemy na konsoli oba bufory.

Rekurencja – argument i wynik przez rejestry

$$\begin{cases} x_0 = 3 \\ x_n = -5x_{n-1} + 7 \end{cases}$$

Argumenty i wynik przekazywane przez rejestry.

W sekcji .data inicjujemy zmienne:

```
LICZBA_WYRAZOW = 4
LICZBA_POCZ = 3
```

Wkładamy do rejestru r9 pierwszy i zarazem jedyny argument funkcji – liczbę wyrazów ciągu.

```
mov $LICZBA_WYRAZOW, %r9 #Licznik pętli


movq $1, %r11
call rekurencyjna
```

Wywołujemy funkcję instrukcją call.

Najpierw sprawdzamy (flaga w r11), czy musimy włożyć cyfrę 3, która jest początkowym wyrazem ciągu – musimy zrobić to tylko raz, przy samym uruchomieniu funkcji, czy już nie – jeśli już to zrobiliśmy i funkcja liczy dalsze wyrazy ciągu.

Po etykiecie „dalej”, gdy sprawdzamy, czy nie powinniśmy już skończyć pętli, wykonujemy pętlę „obliczenia”, która kolejno mnoży nasz poprzedni wyraz (w raxie) razy (-5), a następnie dodaje do niego 7.

Funkcja wywołuje siebie samą – operacje są powtarzane, aż licznik będzie 0.

Ciało funkcji wygląda następująco:

```
rekurencyjna:
cmp $0, %r11
```

```
jne poczatkowy_zabieg


dalej:
cmp $0, %r9
jnz obliczenia
ret


obliczenia:
movq $(-5), %r10
imul %r10
add $7, %rax              #Wynik jest w rax


movq %rax, textout(, %r9, 1)  #Przykładowe wykonanie operacji


dec %r9
call rekurencyjna
ret
```

Początkowy zabieg obejmuje właśnie włożenie cyfry 3 do rejestru.

```
poczatkowy_zabieg:
dec %r11
mov $LICZBA_POCZ, %rax        #Wkladamy 3 - liczbe od ktorej zaczynamy ciag
jmp dalej
```

Rekurencja – argument i wynik przez stos

W sekcji .data deklarujemy zmienne: liczba wyrazów ciągu i cyfra początkowa ciągu.

```
LICZBA_WYRAZOW = 7
LICZBA_POCZ = 3
```

Po etykiecie _start inicjujemy licznik wyrazów ciągu -1 (dopiero w funkcji zostanie przypisana tutaj wartość poprzez stos).

Liczbę wyrazów zapisujemy do rejestru r9.

Ustawiamy flagę r11 – aby tylko na początku wpisać do funkcji rekurencyjną 3 do rejestru rax. Po późniejszych rekurencyjnych wywołaniach, nie chcemy już tego robić.

```
movq $(-1), %rbx

movq $LICZBA_WYRAZOW, %r9    #Liczba wyrazow ciagu w rejestrze r9

movq $1, %r11            #Flaga w r11
```

Wrzucamy na stos liczbę wyrazów ciągi i wywołujemy funkcję.

```
push %r9                 #Wrzucenie liczby wyrazow ciagu na stos

call rekurencyjna        #Wywolanie funkcji rekurencyjnej

add $8, %rsp
```

Na końcu następuje „usunięcie” parametrów ze stosu poprzez przesunięcie wskaźnika rsp, będą one potem nadpisane. Wyniki poszczególnych wyrazów ciągu znajdują się w rejestrze rax.

```
exit:

#---Koniec programu---

mov $SYSEXIT, %rax

mov $EXIT_SUCCESS, %rdi

syscall
```

Tak prezentuje się funkcja rekurencyjna.

```
#===Funkcja rekurencyjna===


rekurencyjna:
```

Tylko na początku chcemy wykonywać instrukcję pobierania wartości ze stosu i wpisywania jej do rejestru rbx.

```
    cmp $(-1), %rbx              #Spr   czy   juz   wyliczono wszystkie wtrazy ciagu

    jne spr_flagi           #Jesli nie, obliczenia


```

```
    push %rbp          #Umieszczenie na stosie poprzedniej
wartości rejestru bazowego

    mov %rsp, %rbp          #Pobranie zawartosci rejestru rsp
(wsk na ost element stosu) do rejestru bazowego

    sub $8, %rsp            #Zwiększenie wskaźnika stosu o 1
komórkę (ominięcie adr powrotu)



    mov 16(%rbp), %rbx      #Zapisanie argumentu ze stosu do
rejestru rbx
```

Sprawdzamy, czy liczymy pierwszy wyraz, czy dalszy.

```
    spr_flagi:

    cmp $0, %r11            #Sprawdzenie,  czy  jest  flaga,
jeśli tak - początkowy zabieg

    je dalej
```

Jeśli pierwszy, wykonujemy zabieg początkowy.

```
    #===Zabieg początkowy===

    #===Usuniecie   flagi,   wpisanie   3   jako   liczbe
rozpoczynajaca ciag===

    #===Zapisanie  argumentu  przekazanego  przez  stos  w
rejestrze rbx===

    dec %r11           #Flaga z 1 na 0

    mov $LICZBA_POCZ, %rax       #Wkladamy  3  -  liczbe  od
ktorej zaczynamy ciag
```

Trzeba zabezpieczyć program przed przypadkiem, gdy ktoś chce jedynie jeden wyraz obliczyć, czyli 3. Wtedy nie należy wykonywać żadnych obliczeń, ale zwrócić 3 zapisaną na początku w rejestrze rax.

```
    cmp $1, %rbx            #Jesli  ktos  chce  tylko  jeden
wyraz ciagu, konczymy wczesniej z wynikiem 3 w rax

    je czyszczenie



    dalej:

    cmp $0, %rbx            #Spr czy juz wyliczono wszystkie
wtrazy ciagu

    jne obliczenia          #Jesli nie, obliczenia
```

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

Obliczanie kolejnych wyrazów ciągu.

```
     obliczenia:
     movq $(-5), %r10         # -5
     imul %r10           # -5 * x_(n-1)
     add $7, %rax             #-5 * x_(n-1) + 7 -> wynik do rax


     dec %rbx            #Obniżenie licznika
call rekurencyjna        #Rekurencyjne wywołanie funkcji - działania na raxie
```

Wnioski

W programie o zliczaniu ciągów zer, należało wziąć pod uwagę przypadek, w którym indeks jest większy od 10 – taką wartość należało zamienić na ASCII, wcześniej dzieląc ją przez 10 i do każdej uzyskanej cyfry dodawać +48, aby zakodować ją w ASCII.

Instrukcje:

```
push %rbp          #Umieszczenie na stosie poprzedniej wartości rejestru bazowego
     mov %rsp, %rbp          #Pobranie zawartosci rejestru rsp (wsk na ost element stosu) do rejestru bazowego
```

należy wykonywać od razu po rozpoczęciu funkcji rekurencyjnej, jeśli są umieszczone w innej etykiecie, możemy napotkać błąd Segmentation fault.

Operacja funkcji rekurencyjnej z wykorzystaniem stosu wymagała użycia wielu etykiet.

Źródła:

Professional Assembly Language, Richard Blum

http://jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl/Architektura-Komputerow/lab/Architektura-52.pdf