Wojciech Adles, 209853 Wrocław, 10.05.17r

ŚR-N-15

Laboratorium Organizacji i Architektury Komputerów

Laboratorium 3:

Funkcje

1. Treść ćwiczenia

- Napisać funkcje rekurencyjną $n_i=n_{i-1}*n_{ni-2}$ (1 pkt)
- Argumenty przekazywane przez stos (1 pkt)
- Argumenty przekazywane przez rejestry (1 pkt)
- Napisać funkcję sprawdzającą czy liczba jest pierwsza (2 pkt)

2. Przebieg ćwiczenia

2.1 Operacje na stosie

Pierwszym zadaniem było napisanie funkcji rekurencyjnej której argumenty będą przekazywane przez stos. Miała ona zwracać wartość wyrazu ciągu o indeksie zadeklarowanym podczas jej wywołania. Aby móc wykonać to zadanie, potrzeba jest znajomość zasad działania stosu w architekturze x86.

Stos jest tutaj zaimplementowany jako struktura w której kolejne elementy mają adresy „mniejsze” od poprzednich, dlatego chcąc odwołać się do elementów znajdujących się „głębiej” w stosie musimy zrobić to za pomocą pośredniego adresowania, dodając do adresu znajdującego się pod rejestrem *%rsp* odpowiedni offset (8 bajtów dla architektury 64-bitowej). Poniżej znajduje się fragment programu służący do inicjalizacji stosu zmiennymi i wywołaniem funkcji *recursive*.

```
.section .data

SYSEXIT = 60

FIRST_NUMBER = -1
SECOND_NUMBER = -2
N_INDEX = 6                             #Index of desired number

.section .text
.globl _start
_start:
        movq $N_INDEX, %rax             #Ni value to the stack
        push %rax
        movq $FIRST_NUMBER, %rax        #N1 value to the stack
        push %rax
        movq $SECOND_NUMBER,  %rax      #N2 value to the stack
        push %rax
        movq $3, %rax                   #N-index of result number
        push %rax
        movq $0, %r15                   #Iteration counter
        call recursive
        movq 8(%rsp), %rbx
exit:
        movq $SYSEXIT, %rax
        syscall                         #Exiting program
```

W powyższym kodzie można zauważyć, że do rejestru *%r15* jest załadowana wartość 0. Będzie ona potrzebna przy obliczaniu adresu zmiennych odłożonych na stosie, który będzie systematycznie rósł z każdą kolejnym wywołaniem funkcji, zapełniany adresami powrotu przez co zmienne na których operujemy będą coraz to „głębiej” w stosie.

2.2 Funkcja rekurencyjna

Zanim przejdę do opisu napisanej prze mnie funkcji warto przypomnieć, że podczas wywołania funkcji w asemblerze instrukcją *call* odkładany jest na stos adres powrotu, do którego przechodzi program wykonując instrukcję *ret* „zdejmując” go jednocześnie ze stosu. Właśnie dlatego dla poniższego kodu, rekurencyjna funkcja *recursive* będzie odkładała na stos adres powrotu przy każdym jej wywołaniu, przez co argumenty funkcji będą znajdowały się pod względnie dalszymi adresami, licząc od wierzchołka stosu. Poniżej znajduje się pozostały fragment kodu, zawierający ciało funkcji pozostały fragment kodu, zawierający ciało funkcji *recursive*.

```
recursive:
        push %rbp
        movq %rsp, %rbp
        movq 32(%rbp,%r15,8), %rax      #N(i-2) in %rax
        movq 24(%rbp,%r15,8), %rbx      #N(i-1) in %rbax
        imul %rbx, %rax                 #Result is now in %rax
        movq %rbx, 32(%rbp,%r15,8)      #Copy N(i-1) to N(i-2)
        movq %rax, 24(%rbp,%r15,8)      #Result is now N(i-1)
        movq 16(%rbp,%r15,8), %rbx      #Copy index of current result in %rax
        cmp %rbx, 40(%rbp,%r15,8)       #Check if thats the final index
        je end
        inc %rbx
        movq %rbx, 16(%rbp,%r15,8)      #Update current index of result
        inc %r15

        movq %rbp, %rsp
        pop %rbp
        call recursive
end:
        cmp $-1,%r15
        je end2
        movq %rbp, %rsp
        pop %rbp
        movq $-1, %r15
end2:
        ret
```

Na początku funkcji zachowujemy wartość rejestru *%rbp* odkładając go na stos i kopiujemy do niego adres na który aktualnie wskazuje stack pointer. Jest to konieczne z racji tego, że wykorzystamy go przy odwoływaniu się do zmiennych umieszczonych na stosie, przez co relatywny adres naszych zmiennych nie będzie zależny od stanu stosu, który może zostać teraz wykorzystany podczas wykonywania się funkcji.

Następnie ładujemy do rejestrów dwa ostatnie wyrazy ciągu i poprzez mnożenie ze znakiem obliczamy jego następny wyraz. Warto tutaj zwrócić uwagę na wykorzystanie adresowania pośredniego przy odwoływaniu się do zmiennych znajdujących się na stosie. Z racji „rośnięcia" stosu przy każdej iteracji funkcji, nasze zmienne będą coraz „dalej" od adresu wskazywanego przez *%rbp*, co rozwiązujemy zwiększając adres w zależności od numeru danej iteracji, przechowywanego w rejestrze *%r15*.

Po wykonaniu mnożenia kopiujemy wyraz n_{i-1} na miejsce n_{i-2} i wstawiamy w jego miejsce otrzymany iloczyn. Następnie sprawdzamy czy indeks otrzymanego wyniku jest równy indeksowi wyrazu ciągu którego szukamy. Jeśli nie to inkrementujemy aktualny indeks wyrazu wyniku, oraz numer iteracji funkcji. Przypisujemy wskaźnikowi stosu wartość jaką miał na początku wywołania funkcji, „zdejmując" następnie ze stosu starą wartość *%rbp* i wykonujemy kolejną iteracje funkcji, wywołując ją w swoim własnym ciele. Jeśli jednak obliczyliśmy interesujący nas wyraz ciągu, przechodzimy do etykiety *end* i tam sprawdzamy czy to pierwsze nasze przejście do niej, jeśli tak (wartość *%rbp* będzie inna niż -1) to wykonujemy operacje przywrócenia początkowego stanu wskaźnika stosu i „zdejmujemy" z niego zachowaną wartość *%rbp*, tym razem kopiując do rejestru *%r15* wartość -1 aby przy kolejnych iteracjach powrotu nie powtarzać tej operacji.

Po zdjęciu ze stosu wszystkich adresów powrotu kopiujemy otrzymany wynik do rejestru *%rbx*

kończąc program.

3. Wnioski

Podczas laboratorium zdążyłem napisać tylko zalążek programu, który następnie dokończyłem w domu. Poznałem zasadę działania stosu, oraz nabyłem umiejętność pisania funkcji w asemblerze. Pozostałych zadań niestety nie udało mi się zrobić, oddając je w następnym terminie doślę zaktualizowaną wersje sprawozdania.