Wrocław, dn. 6 kwietnia 2017

Hanna Grodzicka, 226154

Funkcje

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 obsługi stosu oraz tworzenia i wywoływania funkcji.

Pierwszym zadaniem było napisanie funkcji liczącej wartość zadanego numeru wyrazu z ciągu Fibonacciego. Funkcja miała być rekurencyjna, a do jej wywołania konieczny tylko jeden argument - numer wyrazu.
Zadanie powinno zostać wykonane w dwóch wariantach:

- przekazując argumenty oraz wynik przez stos;
- przekazując argumenty oraz wynik przez rejestry (od r8 do r15).

Drugie zadanie polegało na modyfikacji wcześniej napisanego programu - szyfru Cezara. Algorytm szyfrowania należało umieścić w funkcji, która wymaga dwóch argumentów: znaku do zaszyfrowania oraz klucza szyfrującego. Funkcja powinna zwrócić zaszyfrowany znak. Przekazywanie argumentów jak i wyniku było dowolne - można było je wykonać zarówno przez rejestry jak i stos.

2. Przebieg ćwiczenia

A. Ciąg Fibonacciego

Program pobiera od użytkownika ciąg znaków z wejścia standardowego, zamienia na wartość liczbową, a następnie przekazuje do funkcji jako jej argument. Do wykonania tej części została wykorzystana część kodu z zadania dot. szyfru Cezara.

Po wykonaniu się funkcji, zwracana wartość jest przekodowywana na ciąg znaków ascii, aby wynik dało się wyświetlić. Jest to dokonywane przez dzielenie zawartości rejestru, w którym zwracany jest wynik, przez 10, w pętli. Do reszty z tego dzielenia dodawana jest wartość `'0'` z ascii, aby przekodować cyfrę na znak. Następnie wynikowy bajt wpisywany jest do bufora `tmp`. Pętla jest powtarzana dopóki wynik z dzielenia nie osiągnie zera. Po takiej operacji bufor `tmp` przechowuje liczbę zapisaną od najmłodszego znaku, do najstarszego, dlatego konieczne jest jego przepisanie w odwrotnej kolejności do `textout`.

Główna część programu została napisana na dwa sposoby:

- **przez stos:**

<u>wywołanie funkcji:</u>

```
pushq   %r10            #arg1 - numer wyrazu
pushq   $0              #miejsce na wynik operacji
jmp dalej
# […]
dalej:
call    fibonacci
popq    %rax
addq    $8, %rsp
```

Jeśli wczytanie liczby z klawiatury przebiegnie pomyślnie, to program skacze pod etykietę `dalej`, w której wywoływana jest funkcja. Wynik wykonania się funkcji zapisywany jest na stosie, dlatego następną operacją jest jego zdjęcie z szczytu stosu za pomocą `pop`. Instrukcja `addq $8, %rsp` przesuwa wskaźnik stosu o 8 bajtów do tyłu/w dół/niżej - przechodzi pod element, który obecnie leży na szczycie stosu. W ten sposób element, który leżał na szczycie zostaje zutylizowany tą instrukcją. W rzeczywistości nie dokonują się na nim żadne zmiany, ale przy następnych operacjach, w wyniku których będziemy chcieli odłożyć coś na stos, jego wartość zostanie nadpisana.

funkcja:

Wykorzystana funkcja była inspirowana kodem z języka C:

```
fibonacci(int nr) {
   if(nr == 1 || nr == 2)
       return 1;
   else
       return fibonacci(nr - 1) + fibonacci(nr - 2); }
```

Każde wywołanie funkcji dla wyrazów, które nie są 1 lub 2, skutkuje dwukrotnym wywołaniem rekurencyjnym.

```
fibonacci:
pushq   %rbp
movq    %rsp, %rbp
```

Powyższe instrukcje służą kolejno: do zapisania obecnego miejsca na stosie, w które wskazuje `rbp` (zachowanie jego wartości), do zachowania wskaźnika stosu `rsp` w `rbp`.

Na chwilę obecną stos wygląda tak:

```
# arg1 ($nr)        <-  24(%rbp)
# wynik ($0)        <-  16(%rbp)
# adres powrotny    <-  8(%rbp)
# dawny %rbp        <-  (%rbp) i (%rsp)
```

Adres powrotny został odłożony na stos automatycznie w momencie wywołania komendy `call`. `arg1` to numer wyrazu z ciągu Fibonacciego do wyliczenia.

```
movq    24(%rbp), %rax   #pobranie arg1 ze stosu
cmp     $2, %rax         #if(nr == 2)
je      return1          #goto return1
cmp     $1, %rax         #if(nr == 1)
je      return1          #goto return1
decq    %rax             #nr - 1
```

```
pushq   %rax            #odłożenie argumentu na szczyt stosu
pushq   $0              #miejsce na wynik
```

Wygląd stosu aż do powyższej linijki:

```
# arg1 ($nr)        <-  24(%rbp)
# wynik ($0)        <-  16(%rbp)
# adres powrotny    <-  8(%rbp)
# dawny %rbp        <-  (%rbp)
# arg1 ($nr-1)      <-  -8(%rbp)
# wynik ($0)        <-  -16(%rbp) i (%rsp)
```

W kolejnym wywołaniu funkcji, będzie ona korzystała tylko z dwóch ostatnio odłożonych parametrów.
Pierwsze z wywołań rekurencyjnych:

```
call    fibonacci       #fibonacci(nr - 1)          1ST CALL
```

Po wywołaniu tej funkcji stos wygląda tak, jak przed jej wywołaniem, za wyjątkiem miejsca na `wynik`, w które zostaje wpisany wynik wywołania funkcji dla argumentu, który pod nim leży.
Rezerwacja miejsca na stosie na dwa parametry (po 8 bajtów):

```
subq    $16, %rsp       #przesunięcie wskaźnika stosu
```

Oprócz dwóch parametrów, które są na szczycie stosu, po pierwszym wywołaniu rekurencyjnym, będą umieszczone nad nimi kolejne dwa, do drugiego wywołania. Rezerwowane jest miejsce na liczbę nr-2 oraz na wynik.

```
movq    -8(%rbp), %rax  #nr z 1st call
movq    %rax, -24(%rbp) #nr dla 2nd call
movq    $0, -32(%rbp)   #wynik 2nd call
decq    -24(%rbp)       #nr - 2
```

Liczba nr-1, dla której było pierwsze wywołanie jest kopiowana na szczyt stosu, a następnie dekrementowana. Nad nią jest miejsce na wynik.

Wygląd stosu na ten moment:

```
# arg1 ($nr)        <—  24(%rbp)
# wynik ($0)        <—  16(%rbp)
# adres powrotny    <—  8(%rbp)
# dawny %rbp        <—  (%rbp)
# arg1 ($nr-1)      <—  -8(%rbp)
# wynik             <—  -16(%rbp)
# arg1 ($nr-2)      <—  -24(%rbp)
# wynik ($0)        <—  -32(%rbp) i (%rsp)
```

Po wywołaniu funkcji po raz kolejny, będzie ona korzystała z dwóch parametrów na szczycie stosu.

```
call    fibonacci         #fibonacci(nr - 2)      2ND CALL
```

Po wywołaniu tej linijki stos powinien być taki, jak ostatnio opisany, tylko w miejsce `-32(%rbp)`, powinien zostać umieszczony wynik dla wyrazu nr-2 z ciągu.

```
movq    -16(%rbp), %rax  #wynik z 1st call
addq    -32(%rbp), %rax  #wynik z 2nd call
```

Wyniki obu wywołań zostają zsumowane w rejestrze `rax`.

```
addq    $16, %rsp         #usuwa dwie wartości ze szczytu
                          #stosu przez przesunięcie wskaźnika
```

Wsadzenie wyliczonego wyniku dwóch wywołań funkcji w miejsce na wynik, na stosie. Jest to miejsce nad argumentem, dla którego była wywoływana funkcja na samym początku.

```
movq    %rax, 16(%rbp)
jmp     end_fibonacci
```

Wartość 1 jest zwracana dla pierwszego i drugiego wyrazu ciągu. Skok do tej instrukcji może się odbyć na początku jej działania, gdzie były umieszczone porównania ze skokami warunkowymi.
Ta liczba jest wsadzana w miejsce na wynik na stosie.

```
return1:
movq    $1, 16(%rbp)
```

Na koniec działania funkcji wskaźnik stosu jest ustawiany na miejsce, na które wskazywał po przejściu pod etykietę `fibonacci`. Do rejestru bazowego jest zwracana wartość, którą przechowywał przedtem, a za pomocą instrukcji `ret`, zostaje zdjęty ze stosu adres powrotny, gdzie będą wykonywane dalsze instrukcje.

```
end_fibonacci:
movq    %rbp, %rsp
popq    %rbp
ret
```

Po wyliczeniu wartości docelowej, jedne parametry na stosie, które się na nim ostają, to arg1 i wynik.

- **przez rejestry:**

<u>wywołanie funkcji:</u>
Do wykonania funkcji będą wykorzystywane cztery rejestry.

```
#arg1 w %r10
movq    $2, %r15        #licznik
movq    $1, %r14        #a
movq    $1, %r13        #b

jmp dalej
# […]
dalej:
call    fibonacci
movq    %r13, %rax      #zwracana wartość
```

funkcja:

Do wykonania funkcji posługując się wyłącznie rejestrami została użyta inna funkcja, która jest rekurencją ogonową. Ten typ rekurencji jest szybszy w wykonaniu, a gdyby zastosować ją do poprzedniego wariantu zadania, to nie byłoby groźby przepełnienia stosu.

```
#################
# int fibonacci(int a, int b, int licznik)
# {
#    if(licznik < n)
#       return fibonacci(b, a + b, licznik + 1);
#    else return b;
# }
#
# wywołanie funkcji:    fibonacci(1,1,2)
#
# licznik - zmienna od 2 do n-1
# a - wyraz numer x
# b - wyraz numer x+1
# n - numer wyrazu (arg1)
#
# r15 - licznik; wartość początkowa: 2
# r14 - a;       wartość początkowa: 1
# r13 - b;       wartość początkowa: 1
# r10 - n;       wartość początkowa: podaje użytkownik(const)
#
# wynik zwracany w r13 (b)
#################

fibonacci:
```

W tym miejscu zachowywanie rbp i rsp jest zbędne, ponieważ funkcja nie będzie dokonywała żadnych zmian na stosie.

```
#wyjątki, których funkcja by nie policzyła
cmp     $2, %r10        # if(n == 2)
je      end_fibonacci
cmp     $1, %r10        # if(n == 1)
je      end_fibonacci
```

Dla wyrazu pierwszego i drugiego funkcja od razu kończy swoje wykonanie i w dalszej części kodu wynik jest przekazywany przez rejestr `r13`.

```
cmp     %r10, %r15      # if(licznik < n)
jge     end_fibonacci   # skok <=> (licznik <= n)

movq    %r14, %r12      # kopia zmiennej a (%r14) w %r12
movq    %r13, %r14      # a = b
addq    %r12, %r13      # b = b + a
incq    %r15            # licznik++
```

Wywołanie funkcji dla argumentów, które zostały powyżej wyliczone:

```
call    fibonacci       # return fibonacci(b,a+b,licznik+1);

end_fibonacci:
ret
```

Funkcja ta jest raczej przejrzysta i znacznie prostsza niż poprzednia.

B. Szyfr Cezara

W poprzednio wykonanym programie, część (algorytm) odpowiedzialna za szyfrowanie znaku, została przesunięta na koniec pliku. W miejscu tego algorytmu została wstawiona linijka:

```
call    cezar
```

Przed wywołaniem funkcji w odpowiednich miejscach (po wczytaniu i ewentualnym przekodowaniu na wartość liczbową) na stosie lądują dwa argumenty:

```
# […]
pushq   %r10          #arg1 - klucz
# […]
pushq   %r13          #arg2 - znak
```

Zadaniem funkcji `cezar` jest zaszyfrowanie jednego znaku, więc będzie ona wykonywana tylko raz.

Pod etykietą funkcji są dwie standardowe linijki do zachowania rejestrów stosu. Następnie zostają zdjęte ze stosu dwa wcześniej odłożone argumenty i przeniesione do odpowiednich rejestrów, które są później używane w funkcji.

```
cezar:
pushq   %rbp
movq    %rsp, %rbp

movq    16(%rbp), %rbx          #pobranie znaku (arg2)
movq    24(%rbp), %r10          #pobranie klucza (arg1)
# […]
```

Algorytm szyfrowania nie uległ żadnym zmianom.

Zakończenie funkcji:

```
wypisz:
movq    %rbx, %rax              #wynik działania funkcji
                                #zwracany jest w %rax

movq    %rbp, %rsp
popq    %rbp
ret
```

Jedyną zmianą w pozostałym kodzie (pod wywołaniem funkcji), jest dopisanie czterech linijek, które przekazują zaszyfrowany znak do bufora `textout`.

```
movq    $0, %r14                #licznik textout
movq    %rax, textout(,%r14,1)
incq    %r14
movq    $'\n', textout(,%r14,1)
```

3. Podsumowanie i wnioski

Obie funkcje liczące wyrazy z ciągu Fibonacciego mogą wypisać liczbę (wartość dla zadanego wyrazu) mieszczącą się na maksymalnie 64 bitach. Ta liczba to 2^{64} -1 ≈ 1.84 * 10^{19}. Zatem obie funkcje policzą maksymalnie 189. wyraz ciągu.
Przy wersji ze zwykłą rekurencją i z wykorzystaniem stosu może wystąpić jego przepełnienie. Wersja z wykorzystywaniem rejestrów działa szybciej ze względu na przyjęcie innej funkcji (rekurencji ogonowej).

4. Bibliografia

- J.Barlett *Programming from the ground up*, 2003
- https://pl.wikipedia.org/wiki/Asembler_x86