lab_8 - Instrukcja do ćwiczenia

Teoria:

Zmienne lokalne:

W sytuacji gdy funkcja rekurencyjna korzysta ze zmiennych lokalnych, jedynym miejscem gdzie mogą się one znaleźć jest stos. Tworząc kod w asemblerze programista musi użyć odpowiednich instrukcji, aby zarezerwować na stosie wymaganą wielkość miejsca na zmienne lokalne, a także by bezproblemowo móc korzystać z takich zmiennych. Najprostszym sposobem jest rozszerzenie kodu funkcji o dwa bloki: tzw. **prolog** i **epilog**.

Prolog:

- **push %rbp** - funkcja nie może zmienić zawartości **%rbp**, więc konieczne jest zapamiętanie oryginalnej zawartości na stosie.
- **mov %rsp, %rbp** – rejestr **%rbp** zostanie wykorzystany do adresowania danych znajdujących się na stosie (czasami możliwe jest użycie do tego celu rejestru **%rsp**, ale nie jest zalecane ze względu na jego „chwiejność" (skłonność do zmiany swojej wartości, a co za tym idzie, do zmiany wskazywanego obiektu)).
- **sub size, %rsp** – rezerwacja miejsca na stosie (**size** jest wielkością rezerwowanego obszaru).

Epilog:

- **mov %rbp, %rsp** – przywrócenie rejestrowi **%rsp** wartości jaką miał na początku funkcji. Instrukcja ta z jednej strony „zwalnia" obszar zarezerwowany na zmienne lokalne, z drugiej naprawia ewentualne błędy w położeniu wskaźnika stosu (spowodowane np. niezgodnością liczby instrukcji **push** i **pop**) – z tego względu jest lepszym rozwiązaniem niż instrukcja **add size, %rsp**.
- **pop %rbp** – odtworzenie oryginalnej zawartości **%rbp**.

Odwołanie do zmiennej lokalnej dokonywane jest zwykle poprzez tryb adresowania pośredniego względem zawartości rejestru **%rbp** czyli np. **mov %rax, -8(%rbp)** – przesunięcie jest zawsze ujemne (zmienne lokalne są na stosie poniżej miejsca wskazywanego przez **%rbp** (pod adresami mniejszymi niż zawartość **%rbp**), wartość przesunięcia zależy od liczby zmiennych lokalnych i ich rozmiarów (**byte**, **word**, **long**, itd.) – do wykorzystania jest obszar od **–size(%rbp)** do **-1(%rbp)**.

Praktyka (lab_8a.c+lab_8a_asm.s oraz lab_8b.c+lab_8b_asm.s):

Działania:

1. Testujemy działanie funkcji **factc** (język **C**) oraz **facta** (asembler) liczących silnię dla argumentów będących liczbami nieujemnymi. Program główny zawiera wywołania obu funkcji, co pozwala porównać uzyskiwane rezultaty. Obliczenia prowadzone są na danych 32-bitowych.
2. CL (Compile&Link) – polecenie: **gcc –o lab_8a lab_8a.c lab_8a_asm.s**
3. R (Run) – polecenie: **./lab_8a**
4. Pojawiają się wyniki dla argumentów od **1** do **6** – obie funkcje dają się identyczne wyniki.
5. Zmieniamy górną granicę licznika pętli w funkcji **main** na **30**:

```
for( i = 1; i <= 30; i++ )
```

6. CLR
7. Dla argumentu równego **17** wartości silni są ujemne, co świadczy o przepełnieniu rejestru w trakcie mnożenia. Wcześniejsze wartości też świadczą o przepełnieniu (brak zera na końcu liczby, wartości dla argumentu **i+1** mniejsze niż dla **i**).
8. Modyfikujemy kod tak, aby obliczenia były prowadzone na danych 64-bitowych.
9. Kod w **C** należy zmienić w czterech miejscach:

```
long factc( unsigned int k )

long facta( unsigned int k );

printf( "FactC(%d) = %ld\n", i, factc(i) );

printf( "FactA(%d) = %ld\n", i, facta(i) );
```

10. Modyfikacja kodu w pliku **lab_8a_asm.s** polega na użyciu wszędzie rejestrów 64-bitowych:

```
mov $1, %rax          # result
cmp %rax, %rdi        # rdi <= 1 ?
dec %rdi              # new parameter (k-1)
mul %rdi              # k! = (k-1)! * k
```

11. CLR
12. Okazuje się, że tym razem przepełnienie pojawia się dla argumentu równego **21** (wartość ujemna). Dokładniejsze porównanie wyników pozwala na zauważenie, że w wersji 32-bitowej przepełnienie pojawia się już dla argumentu równego **13** (ostatnia cyfra to **4** zamiast **0**).

13. Przechodzimy do programu **lab_8b** – celem jest przetestowanie funkcji wyznaczających wartości kolejnych wyrazów ciągu Fibonacciego. Program składa się z dwóch modułów: w **C** (funkcje **fibc** i **main**) i w asemblerze (funkcja **fiba**).
14. CL (Compile&Link) – polecenie: **gcc –o lab_8b lab_8b.c lab_8b_asm.s**
15. R (Run) – polecenie: **./lab_8b**
16. Pojawiają się wyniki dla argumentów od **0** do **10** – obie funkcje dają się identyczne wyniki.
17. Zmieniamy górną granicę licznika pętli w funkcji **main** na **50**:

```
for( i = 1; i <= 50; i++ )
```

18. CLR
19. Czekamy na wyniki – w zależności od sprzętu może to zająć kilka do kilkunastu minut!
20. Dla argumentu równego **47** wartości są ujemne, co świadczy o przepełnieniu rejestru w trakcie dodawania.
21. Modyfikujemy kod tak, aby obliczenia były prowadzone na danych 64-bitowych.
22. Kod w **C** należy zmienić w czterech miejscach:

```
long fibc( unsigned int k ) {
long fiba( unsigned int k );
printf( "FibC( %d ) = %ld\n", i, fibc( i ) );
printf( "FibA( %d ) = %ld\n", i, fiba( i ) );
```

23. Modyfikacja kodu w pliku **lab_8a_asm.s** polega na użyciu wszędzie rejestrów 64-bitowych:

```
cmp $0, %rdi          # parameter (k) == 0 ?
cmp $1, %rdi          # k == 1 ?
sub $2,%rdi           # new parameter k-2
mov %rax,-8(%rbp)     # store result in local variable
dec %rdi              # new parameter k-1
add -8(%rbp),%rax     # %rax += local variable
mov $0, %rax          #  k == 0
mov $1, %rax          #  k == 1
```

24. CLR
25. Ponownie czekamy na wyniki – w zależności od sprzętu może to zająć kilka do kilkunastu minut!
26. Okazuje się, że tym razem przepełnienie się nie pojawia (dla przetestowanych argumentów) – tak naprawdę, to przepełnienie pojawi się dla argumentu równego **93**:

```
FibC( 93 ) = -6246583658587674878
FibA( 93 ) = -6246583658587674878
```

ale lepiej tego nie sprawdzać – można spróbować, ale można też nie doczekać.

27. Sprawdzamy przyczynę tak wolnego działania programu.
28. Usuwamy w kodzie C komentarze poprzedzające użycie zmiennej **nest_lev**:

```
//    long nest_lev;
//    nest_lev++;
//    nest_lev = 0;
//    printf( "Nesting level = %ld\n", nest_lev );
```

29. Pozwoli to na sprawdzenie, jaka liczba wywołań funkcji **fibc** jest konieczna do wyznaczenia wartości odpowiedniego wyrazu ciągu (poziom zagnieżdżenia wywołań funkcji).
30. Aby skrócić czas obliczeń, zmieniamy górną granicę licznika pętli w funkcji **main** na **40**:

```
for( i = 1; i <= 40; i++ )
```

31. CLR
32. Widać, że wraz ze wzrostem argumentu liczba wywołań funkcji (poziom zagnieżdżenia) gwałtownie rośnie:

```
FibC( 39 ) = 63245986
FibA( 39 ) = 63245986
Nesting level = 204 668 309
FibC( 40 ) = 102334155
FibA( 40 ) = 102334155
Nesting level = 331 160 281
```

33. Zmiana argumentu z **39** na **40** wiąże się z koniecznością wykonania prawie **130** mln dodatkowych wywołań funkcji – zakładając, że każde wywołanie zajmuje np. **10** ns, to przyrost czasu obliczeń wynosi ~**1,3** sekundy!
34. Ciesz się z tego, co osiągnęłaś/osiągnąłeś – i pomyśl, ile mógłbyś/mógłbyś osiągnąć mając szybszy komputer… ;-)