lab_10 - Instrukcja do ćwiczenia

Teoria:

Obliczenia zmiennoprzecinkowe:

Obliczenia zmiennoprzecinkowe mogą być realizowane w dwóch jednostkach procesora: w tzw. koprocesorze matematycznym **x87** (**FPU** – **Floating Point Unit**) oraz w jednostce **SSE** (**Streaming SIMD Extensions**). **FPU** zawiera **8** rejestrów 80-bitowych, zorganizowanych w formie stosu – nazwa rejestru odnosi się do jego położenia względem wierzchołka stosu (**ST(0)** lub **ST0** lub **ST** to rejestr znajdujący się w danym momencie na szczycie stosu, **ST(1)** to rejestr znajdujący się poziom niżej, itd.). Rejestr może zawierać daną **.float**/**.single** (**32** bity), daną .**double** (**64** bity) lub daną **.tfloat** (**80** bitów) – dane w postaci liczb całkowitych wymagają jawnej konwersji (poprzez użycie odpowiedniej instrukcji, wskazującej że argument jest liczbą całkowitą). Stałe są dostępne tylko poprzez użycie dedykowanych instrukcji (**FLDZ** – **0.0**, **FLD1** – **1.0**, **FLDPI** – **pi**, itp.) – inne wartości mogą się pojawić tylko w formie zmiennych. Jednostka **SSE** udostępnia **8**/**16** rejestrów (w zależności od generacji procesora) o nazwach **xmm0**..**xmm7**/**15** – o rozmiarze **128** bitów. W rejestrze można umieścić więcej niż jedną daną (**2** liczby **double**, **4** liczby **float**/**long**, …, **16** liczb **byte**) - możliwe jest też wykonanie operacji jednocześnie na wszystkich danych zawartych w rejestrze co pozwala na przetwarzanie danych w sposób równoległy.

Ponieważ obie jednostki pojawiły się na różnych etapach rozwoju architektury procesora, to cechują się różną filozofią działania i korzystają z różnych zbiorów instrukcji. W dużym uproszczeniu: jednostka **x87** powinna być używana tylko w zastosowaniach wymagających największej możliwej precyzji obliczeń – wszystkie inne zadania powinny być realizowane z użyciem jednostki **SSE** lub **AVX** (będącej rozwinięciem SSE) – w nowych procesorach.

Przekazywanie parametrów i zwracanie rezultatu:

Argumenty w postaci liczb zmiennoprzecinkowych (maksymalnie **8**) są przekazywane do funkcji w rejestrach **xmm0**..**xmm7** – te, które się nie zmieściły (powyżej **8**) z użyciem stosu.

Funkcja zwracająca rezultat w postaci liczby zmiennoprzecinkowej musi do tego celu użyć rejestru **xmm0** (ewentualnie **xmm0** i **xmm1** w celu zwrócenia pary wartości – np. liczby zespolonej).

Metoda Newtona:

W programie wykorzystana została metoda Newtona (Newtona-Raphsona) do iteracyjnego wyznaczania wartości pierwiastka kwadratowego z danej liczby. Szczegóły metody można znaleźć np. tutaj:

https://pl.wikipedia.org/wiki/Metoda_Newtona

W programie wykorzystano następujący wzór iteracyjny:

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right) \quad gdzie\ x_0 = a$$

Praktyka (lab_10a.s, lab_10b.c+lab_10b.c, lab_10c.s):

Działania:

1. Testujemy działanie programu lab_10a liczącego pierwiastki kwadratowe dla liczb całkowitych z zakresu 1..10 trzema sposobami: poprzez użycie dedykowanej instrukcji jednostki x87 (FPU), poprzez użycie dedykowanej instrukcji jednostki SSE oraz poprzez zastosowanie algorytmu iteracyjnego realizowanego w jednostce x87.
2. CL (Compile&Link) – polecenie: **gcc –no-pie –lm –o lab_10a lab_10a.s**
3. R (Run) – polecenie: **./lab_10a**
4. Pojawiają się wyniki dla argumentów od **1** do **10** – wszystkie metody dają identyczne rezultaty.
5. Chcemy sprawdzić jak szybkozbieżna jest metoda Newtona (ile iteracji jest niezbędnych do uzyskania dokładnego wyniku).
6. W obszarze danych dodajemy następujące deklaracje:

```
cntr:
      .long           0
cnt_fmt:
      .string         "Iterations = %d\n "
```

7. W istniejącym kodzie (na czarno) dodajemy następujące instrukcje (na czerwono):

```
      FLDL x     # first approximation (a0) -> ST(0)
      movl $0, cntr
iter:
      incl cntr
      FLDL x     # function argument -> ST(0), ak in ST(1)
      FDIV %ST(1), %ST(0) # ST(0)/ST(1) -> ST(0)     x/ak
```

8. oraz – trochę niżej:

```
      FLDL sqr_c          # load & display second result
      FSTPL     y
      call disp           # display x, y
      mov  cntr, %rsi
```

```
mov  $cnt_fmt, %rdi
mov  $0, %al
call printf
incl i              # next argument
```

9. CL (Compile&Link) – polecenie: **gcc –no-pie –lm –o lab_10a lab_10a.s**
10. R (Run) – polecenie: **./lab_10a**
11. Pojawiają się wyniki dla argumentów od **1** do **10** – okazuje się, że dla nich maksymalna liczba iteracji to 8, co oznacza, że metoda Newtona jest metodą szybkozbieżną.
12. Przechodzimy do programu **lab_10b** składającego się z dwóch modułów: **lab_10b.c** (program główny w języku **C** zawierający wywołania trzech funkcji) oraz **lab_10b.s** (kod trzech funkcji w asemblerze). Działanie programu jest podobne do wcześniejszego, ale tym razem obliczenia są odseparowane od pozostałych operacji – celem programu jest sprawdzenie przekazywania parametrów do funkcji oraz zwracania rezultatu.
13. CL (Compile&Link) – polecenie: **gcc –no-pie –lm –o lab_10b lab_10b.c lab_10b.s**
14. R (Run) – polecenie: **./lab_10b**
15. Pojawiają się wyniki dla argumentów od **1** do **10** – ponownie wszystkie metody dają identyczne rezultaty.
16. Przechodzimy do programu **lab_10c** składającego się z jednego modułu źródłowego - **lab_10c.s** zawiera kod ilustrujący wyznaczanie pierwiastków (rzeczywistych) równania kwadratowego o współczynnikach a, b, c (wartości zapisane w kodzie) z wykorzystaniem jednostki x87 (FPU). Kod podzielony jest na fragmenty realizujące konkretne zadania: wyświetlenie współczynników równania, policzenie delty, wyświetlenie wartości delty, reakcja na jej wartość (delta < 0, delta = 0, delta > 0) oraz policzenie i wyświetlenie wartości pierwiastków (o ile istnieją).
17. dane w programieprogram główny w języku **C** zawierający wywołania trzech funkcji) oraz **lab_10b.s** (kod trzech funkcji w asemblerze). Działanie programu jest podobne do wcześniejszego, ale tym razem obliczenia są odseparowane od pozostałych operacji – celem programu jest sprawdzenie przekazywania parametrów do funkcji oraz zwracania rezultatu.
18. CL (Compile&Link) – polecenie: **gcc –no-pie –lm –o lab_10c lab_10c.s**
19. R (Run) – polecenie: **./lab_10c**
20. Pojawia się komunikat o naruszeniu ochrony pamięci:

```
buba@buba-pc:~/asm/l10$ ./lab_10c
Segmentation fault (core dumped)
```

21. Przyczyną jest użycie funkcji **printf** przy stosie nie wyrównanym do wielokrotności **16** bajtów (stos jest wyrównany przed wywołaniem funkcji **main** – po jej wywołaniu na stosie pojawia się adres powrotu (**8** bajtów). Konieczne jest wyrównanie stosu wewnątrz funkcji **main** – dokonujemy tego przez usunięcie znaków komentarza w następujących liniach kodu:

```
#     sub  $8, %rsp
```

oraz

```
#     add  $8, %rsp
```

W kodzie programu **lab_10a** nie trzeba było tego robić, bo stos był wyrównany: w funkcji **main** wywoływana była funkcja **disp** (dodatkowe **8** bajtów adresu na stosie), a dopiero w funkcji **disp** następowało wywołanie funkcji **printf**.

22. CLR
23. Program działa już poprawnie:

```
buba@buba-pc:~/asm/l10$ ./lab_10c
Coefficients of equation:
A = 1.000000  B = -1.000000  C = -2.000000
Delta = 9.000000
Two roots
Roots of equation:
X1 = -1.000000  X2 = 2.000000
```

24. Testujemy działanie programu na kilku innych zestawach współczynników (pamiętając, że **a** musi być różne od **0**).
25. Spoczywamy na laurach bądź Laurach - w zależności od warunków, preferencji, itp.