lab_11 - Instrukcja do ćwiczenia

Teoria:

Formuła Leibniza:

W programie wykorzystana została formuła Leibniza do iteracyjnego wyznaczania wartości Π. Szczegóły można znaleźć np. tutaj:

https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80

W programie wykorzystano następujący wzór iteracyjny:

$$\Pi = \frac{4}{1} + \frac{-4}{3} + \frac{4}{5} + \frac{-4}{7} + \cdots$$

Obliczenia są realizowane w czterech funkcjach napisanych w języku **C** (**fun_cf1**, **fun_cf2**, **fun_cd1**, **fun_cd2**) oraz dwóch napisanych w asemblerze (**fun_a** i **fun_b**). Różnice pomiędzy funkcjami w **C** sprowadzają się do wykorzystywanych danych (**fun_cf1** i **fun_cf2** – **float**, **fun_cd1** i **fun_cd2** - **double**) oraz do sposobu liczenia (**fun_cf1** i **fun_cd_1** – dodawanie do sumy jednego ułamka w każdym obiegu pętli, **fun_cf2** i **fun_cd2** – odrębne sumowanie ułamków dodatnich i ujemnych i połączenie obu sum na końcu obliczeń).

Funkcje **fun_a** i **fun_b** wykorzystują mechanizm **SSE** do zrównoleglenia obliczeń – w pierwszym przypadku liczone i sumowane są jednocześnie dwa ułamki (dane typu **double**), w drugim obliczenia są prowadzone na danych **float**, co pozwala na równoległe liczenie i sumowanie czterech ułamków.

CRC:

Program crctest zawiera trzy funkcje wyznaczające wartości **CRC** – wykorzystywanym algorytmem jest **CRC-32C**:

https://en.wikipedia.org/wiki/Cyclic_redundancy_check

Funkcje **crc_c** (w **C**) oraz **crc_a** (**asm**) wymagają tablicy kodów **CRC32** – została ona zdefiniowana na początku modułu napisanego w **C**. Trzecia z funkcji (**crc_b**) korzysta z dedykowanej instrukcji procesora.

Wartość **CRC** jest liczona dla porcji danych o rozmiarze **32KB** – aby uzyskać miarodajne oszacowanie szybkości działania poszczególnych funkcji, proces liczenia powtarzany jest **100000** razy (wartość stałej **REPETITIONS**).

Praktyka (lab_11.c+lab_11.s, crctest.c+crc.s):

Działania:

1. Testujemy działanie programu lab_11 liczącego przybliżenia wartości Π z użyciem sześciu funkcji: czterech w C i dwóch w asemblerze.
2. CL (Compile&Link) – polecenie: **gcc –no-pie –lm –o lab_11 lab_11.c lab_11.s**
3. R (Run) – polecenie: **./lab_11**
4. Pojawiają się wyniki dla liczby iteracji (liczby uwzględnionych w sumie składników) zmieniającej się od **2**, poprzez **4**, **8**, **16**, itd., aż do **65536**. Wszystkie funkcje dają zbliżone rezultaty, różnice w czasach wykonania można łatwo uzasadnić. Problemem jest mała dokładność przybliżenia – wszystkie metody zapewniają jedynie **5** cyfr znaczących.
5. Zmieniamy wartości stałych **BASE** (z **2.0** na **10.0**) oraz **LOG_OF_REPETITIONS** (z **16** na **10**) – w ten sposób maksymalna liczba uwzględnionych w sumach ułamków zmieni się z $\mathbf{2^{16}}$ na $\mathbf{10^{10}}$.
6. CL (Compile&Link) – polecenie: **gcc –no-pie –lm –o lab_11 lab_11.c lab_11.s**
7. R (Run) – polecenie: **./lab_11**
8. Pojawiają się wyniki dla liczby iteracji (liczby uwzględnionych w sumie składników) zmieniającej się od **10**, poprzez **100**, **1000**, **10000**, itd., aż do **10000000000**. Niektóre z wyników znacznie odbiegają od wartości dokładnych – końcowe rezultaty dla poszczególnych metod (funkcji) wyglądają następująco:

```
[CD1]   10000000000 iterations - value = 3.1415926548834582
Time = 65157.0690 ms
[CD2]   10000000000 iterations - value = 3.1415926548480332
Time = 64553.4550 ms
[CF1]   10000000000 iterations - value = 3.14159679412841797
Time = 29987.5380 ms
[CF2]   10000000000 iterations - value = 2.45036125183105469
Time = 29626.0130 ms
[ASM]   10000000000 iterations - value = 3.1415926548480332
Time = 32934.1230 ms
[ASM2]  10000000000 iterations - value = 2.98876094818115234
Time =  7488.4670 ms
```

9. Sensowne (i oczekiwane) rezultaty zapewniają tylko funkcje operujące na danych typu **double**. Typ **float** nie zapewnia wystarczającej dokładności obliczeń – metoda **CF1** jest pewnym wyjątkiem od reguły, bo daje wynik w miarę poprawny. Wynika to ze sposobu działania – sumowanie kolejno ułamków dodatnich i ujemnych pozwala na korygowanie sumy aż do momentu, w którym wartości ułamków są praktycznie równe **0**. Metody wykorzystujące dane **float** i sumujące niezależnie ułamki dodatnie i ujemne (**CF2** i **ASM2**), nie zapewniają takiej korekty i dają zdecydowanie gorsze rezultaty – widać wyraźnie, że sensowne wyniki pojawiają się dla stosunkowo niewielkiej liczby iteracji, a później się znacząco pogarszają i w końcu przestają się zmieniać.
10. Szczegółowa analiza wyników dla **ASM2** (podobnie dla **CD1** i **CD2**) pozwala na zauważenie, że zwiększenie liczby iteracji **10** razy skutkuje poprawieniem dokładności przybliżenia i uzyskaniem kolejnej cyfry znaczącej:

```
[ASM]            10 iterations - value = 3.05840276592733229
```

```
[ASM]           100 iterations - value = 3.13178896757345360
[ASM]          1000 iterations - value = 3.14059464984626846
[ASM]         10000 iterations - value = 3.14149267358604867
[ASM]        100000 iterations - value = 3.14158265378970292
[ASM]       1000000 iterations - value = 3.14159165359170700
[ASM]      10000000 iterations - value = 3.14159255358582712
[ASM]     100000000 iterations - value = 3.14159264358449164
[ASM]    1000000000 iterations - value = 3.14159265258555465
[ASM]   10000000000 iterations - value = 3.14159265348480332
Time = 32934.1230 ms
```

11. Podsumowując: użycie danych **float** skutkuje szybszym działaniem kodu, ale w niektórych sytuacjach mała dokładność może stanowić istotny problem. Typ **double** zapewnia wymaganą dokładność, ale czas obliczeń się wydłuża. Sensownym rozwiązaniem jest wybór metody **ASM** – obliczenia z użyciem danych **double**, ale zrównoleglenie operacji pozwala na znaczące skrócenie czasu obliczeń w porównaniu do metod **CD1** i **CD2**.
12. Przechodzimy do programu **crctest** składającego się z dwóch modułów: **crctest.c** (program główny w języku **C** zawierający wywołania trzech funkcji i kod jednej z nich) oraz **crc.s** (kod dwóch funkcji w asemblerze). Celem jest porównanie szybkości działania różnych sposobów obliczania wartości **CRC**.
13. CL (Compile&Link) – polecenie: **gcc –no-pie –o crctest crctest.c crc.s**
14. R (Run) – polecenie: **./crctest**
15. Przykładowe wyniki wyglądają następująco:

```
buba@buba-pc:~/asm/l11$ gcc -no-pie -o crctest crctest.c crc.s
buba@buba-pc:~/asm/l11$ ./crctest
In C    - CRC = 57FD18E5, time = 16.536 seconds
In ASM1 - CRC = 57FD18E5, time =  7.307
In ASM2 - CRC = 57FD18E5, time =  2.596
```

16. Wszystkie metody dają te same wyniki, ale czasy wykonania są różne: najszybsza metoda to wykorzystanie w kodzie asemblerowym dedykowanej instrukcji, najwolniejsza to programowa realizacja w **C**.
17. Zastosowanie agresywnej optymalizacji kodu w trakcie kompilacji zmienia wyniki:

```
buba@buba-pc:~/asm/l11$ gcc -O3 -no-pie -o crctest crctest.c crc.s
buba@buba-pc:~/asm/l11$ ./crctest
In C    - CRC = 57FD18E5, time =  7.042 seconds
In ASM1 - CRC = 57FD18E5, time =  7.135
In ASM2 - CRC = 57FD18E5, time =  2.616
```

18. Kodu w **C** już bardziej przyspieszyć się nie da (w miarę prosty sposób), natomiast można zoptymalizować kod **ASM1** oraz **ASM2** (w niewielkim zakresie) – jednak nawet bez dodatkowych zabiegów, to metoda **ASM2** zapewnia najszybsze działanie.
19. Cieszymy się, bo dotarliśmy do końca – więcej zadań nie będzie i można zająć się czymś ciekawszym/bardziej pożytecznym/bardziej sensownym[1].

[1] niepotrzebne skreślić