# lab_10 - Instrukcja do ćwiczenia

**Teoria:**

http://galaxy.agh.edu.pl/~amrozek/AK/lab10.pdf

**Pomiar czasu wykonania:**

Pomiar czasu wykonania pewnego fragmentu kodu (np. funkcji) dokonywany jest poprzez odpowiednie użycie dwóch funkcji zawartych w pliku **eval_time.c**: **init_time** i **read_time**. Funkcja **init_time** powinna być wywołana bezpośrednio przed kodem, którego czas wykonania chcemy mierzyć, zaś funkcja **read_time** bezpośrednio po nim. Przykładowe użycie obu funkcji wygląda następująco:

```
#include <stdio.h>
#include "eval_time.h"
void main(void)
{
      double times[3];

      init_time();
      some_function();
      read_time( times );
      printf("T0=%lf T1=%lf T2=%lf\n",times[0],times[1], times[2] );
}
```

Czas **T0** jest czasem spędzonym na poziomie systemu operacyjnego (realizacja funkcji systemowych), **T1** to czas spędzony na poziomie użytkownika, zaś **T2** jest łącznym czasem jaki upłynął od rozpoczęcia pomiaru do jego zakończenia (tzw. czas zegarowy).

$$\mathbf{T0 + T1 \leq T2}$$

Pojedynczy pomiar obarczony jest znacznym błędem ze względu na działanie w systemie wielozadaniowym, więc aby uzyskać miarodajne wyniki należy:

- powtórzyć pomiar kilka-kilkanaście razy,
- mierzyć wyłącznie kod nie zawierający wywołań funkcji systemowych,
- wykorzystywać jedynie czas **T1**.

**Optymalizacja kodu:**

Wykorzystywany kompilator (**gcc**) pozwala na znaczącą optymalizację generowanego kodu, ale uzyskane rezultaty (czasy działania) zależą w znacznym stopniu od sprzętu (generacji procesora, częstotliwości zegara, wielkości pamięci RAM, wielkości pamięci cache, itp.). Poziom optymalizacji ustalany jest na etapie kompilacji kodu źródłowego przy pomocy opcji **Ox**, gdzie **x** jest liczbą z zakresu **[0..3]** – im większa wartość, tym wyższy poziom optymalizacji (**ale nie musi to przekładać się to na szybsze działanie kodu!**). Poziom **0** oznacza brak optymalizacji.

**Formuła Leibniza:**

Formuła Leibniza (https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80) pozwala na wyznaczenie przybliżonej wartości liczby pi przez wyznaczenie sumy skończonej liczby wyrazów:

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

Obliczenia są realizowane w dwóch funkcjach napisanych w języku **C** (**fun_cf**, **fun_cd**) oraz trzech napisanych w asemblerze (**fun_x87d, fun_x87ld** i **fun_ssed**). Różnice pomiędzy funkcjami w **C** sprowadzają się do wykorzystywanych danych (**fun_cf** – **float**, **fun_cd** - **double**).

Funkcja **fun_ssed** wykorzystuje mechanizm **SSE** do zrównoleglenia obliczeń – liczone i sumowane są jednocześnie dwa ułamki (dane typu **double**).

Funkcje **fun_x87d** oraz **fun_x87ld** używają do obliczeń jednostkę **x87** (koprocesor matematyczny) – różnice pomiędzy nimi sprowadzają się do wielkości danych na których wykonywane są obliczenia (**fun_x87d** – **double**, **fun_x87ld** – **double extended**) i związanej z tym precyzji obliczeń.

Ze względów praktycznych warto nieco zmodyfikować wykorzystywany wzór:

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

Pozwoli to na zastosowanie w każdym kroku operacji dodawania (a nie na przemian dodawania i odejmowania. Kolejny wyraz ma zmieniony znak licznika i zwiększoną o **2** wartość mianownika. Ułamki są wyznaczane i sumowane w pętli – przed jej rozpoczęciem celowe jest umieszczenie w rejestrach jednostki **x87** wartości przydatnych w procesie obliczeniowym. Są to: **0** – początkowa wartość sumy, **4** – licznik ułamka, **2** – wartość o którą zwiększany będzie mianownik oraz **1** – początkowy mianownik.

**Praktyka (lab_10a.c, lab_10b.c i pozostałe pliki):**

**Działania:**

1. Przygotowujemy kod do mierzenia czasów wykonania zawarty w eval_time.c.
2. C (Compile) – polecenie: **gcc –O3 –c eval_time.c**
3. Testujemy działanie programu lab_10a.c – dokonuje on pomiarów czasów wykonania trzech różnych fragmentów kodu: zawierającego wywołanie funkcji pozwalającej na

wprowadzenie danych prze użytkownika (scanf), zawierającego wielokrotne wywołanie funkcji zapisującej dane do pliku (fprintf) oraz wielokrotnie powtarzanej w pętli operacji arytmetycznej.

4. CL (Compile, Link) – polecenie: **gcc –O0 –o lab_10a lab_10a.c eval_time.o**
5. R (Run) – polecenie: **./lab_10a**
6. Przykladowe efekty uzyskane po uruchomieniu wyglądają następująco:

```
buba@buba-pc:~/AK/l10$ ./lab_10a
Input: 1
Blocking I/O:     T0 = 0.000000  T1= 0.000191  T2 = 1.446269
Non-blocking I/O: T0 = 0.927476  T1= 1.684605  T2 = 2.612216
Arithmetic:       T0 = 0.000000  T1= 9.018193  T2 = 9.018360
```

7. Kilkukrotne uruchomienie programu skutkuje uzyskaniem różniących się wyników:

```
Input: 1
Blocking I/O:     T0 = 0.000000  T1= 0.000191  T2 = 1.446269
Non-blocking I/O: T0 = 0.927476  T1= 1.684605  T2 = 2.612216
Arithmetic:       T0 = 0.000000  T1= 9.018193  T2 = 9.018360
Input: 2
Blocking I/O:     T0 = 0.000000  T1= 0.000187  T2 = 1.798585
Non-blocking I/O: T0 = 0.955156  T1= 1.591383  T2 = 2.547074
Arithmetic:       T0 = 0.000000  T1= 9.243552  T2 = 9.243678
Input: 3
Blocking I/O:     T0 = 0.000000  T1= 0.000183  T2 = 1.197795
Non-blocking I/O: T0 = 0.974407  T1= 1.607752  T2 = 2.591405
Arithmetic:       T0 = 0.018171  T1= 9.426365  T2 = 9.444710
Input: 4
Blocking I/O:     T0 = 0.000000  T1= 0.000183  T2 = 1.766477
Non-blocking I/O: T0 = 0.992490  T1= 1.575230  T2 = 2.567763
Arithmetic:       T0 = 0.000000  T1= 9.153416  T2 = 9.153516
```

8. Rozrzut wartości **T2** w pomiarach kodu zawierającego wywołanie funkcji **scanf** związany jest z czasem reakcji użytkownika, czasy **T0** i **T1** są mało przydatne (możliwy duży błąd pomiaru, bo funkcja została wywołana tylko raz). Największą wiarygodnością cechuje się pomiar czasu **T1** (i tylko na ten wynik możemy wpłynąć poprzez optymalizację kodu) dla operacji nie zawierających wywołań funkcji systemowych, choć i tutaj pojawiają się czasem odchylenia (trzecie uruchomienie i wyniki dla operacji arytmetycznych). Oznacza to celowość wielokrotnego pomiaru i wyznaczenie wartości uśrednionej po wcześniejszym odrzuceniu pomiarów znacznie odbiegających od pozostałych.
9. W sytuacji gdy interesuje nas tylko czas działania całego programu, to możemy użyć polecenia time:

```
buba@buba-pc:~/AK/l10$ time ./lab_10a
Input: 0
Blocking I/O:     T0 = 0.000000  T1= 0.000188  T2 = 2.086485
Non-blocking I/O: T0 = 0.060074  T1= 1.672568  T2 = 1.732665
Arithmetic:       T0 = 0.000000  T1= 9.535385  T2 = 9.535479
real    0m13,356s
user    0m11,210s
sys     0m0,060s
```

Czas real to odpowiednik T2, user to T1, a sys to czas T0.

10. Przechodzimy do programu **lab_10b.c** – służy on (wraz z plikami **x87.s** oraz **sse.s**) do wyznaczania przybliżonej wartości liczby $\Pi$ na podstawie formuły Leibniza. Odchyłka od dokładnej wartości zależy od precyzji obliczeń oraz od liczby iteracji (liczby składników/ułamków uwzględnionych w sumie).
11. CL (Compile&Link) – polecenie: **gcc -no-pie -o lab_10b lab_10b.c x87.s sse.s eval_time.o -lm**
12. R (Run) – polecenie: **./lab_10b**
13. 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.
14. 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 $\mathbf{10^{10}}$.
15. CL (Compile&Link) – polecenie: **gcc -no-pie -o lab_10b lab_10b.c x87.s sse.s eval_time.o -lm**
16. R (Run) – polecenie: **./lab_10b**
17. 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:

```
[CF]     10000000000 iterations - value = 3.14159679412841797
Time =   29945.0640 ms
[CD]     10000000000 iterations - value = 3.14159265348834582
Time =   64469.8590 ms
[x87D]   10000000000 iterations - value = 3.14159265348834582
Time =   64470.0030 ms
[x87LD]  10000000000 iterations - value = 3.14159265348979935
Time =   70326.9820 ms
[SSED]   10000000000 iterations - value = 3.14159265348480332
Time =   32872.7580 ms
```

18. Sensowne (i oczekiwane) rezultaty zapewniają tylko funkcje operujące na danych typu **double i long double (double extended** – 80 bitów w jednostce x87**)**. Typ **float** nie zapewnia wystarczającej dokładności obliczeń – metoda **CF** jest (do pewnej liczby iteracji) 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** – widać wyraźnie, że sensowne wyniki pojawiają się dla stosunkowo niewielkiej liczby iteracji, a później się znacząco pogarszają i w końcu przestają się zmieniać.
19. Szczegółowa analiza wyników dla **SSED** (podobnie dla **CD**) pozwala na zauważenie, że zwiększenie liczby iteracji **10** razy skutkuje poprawieniem dokładności przybliżenia i uzyskaniem kolejnej cyfry znaczącej:

```
[SSED]          10 iterations - value = 3.05840276592733229
[SSED]         100 iterations - value = 3.13178896757345360
[SSED]        1000 iterations - value = 3.14059464984626846
[SSED]       10000 iterations - value = 3.14149267358604867
[SSED]      100000 iterations - value = 3.14158265378970292
[SSED]     1000000 iterations - value = 3.14159165359170700
[SSED]    10000000 iterations - value = 3.14159255358582712
[SSED]   100000000 iterations - value = 3.14159264358449164
[SSED]  1000000000 iterations - value = 3.14159265258555465
[SSED] 10000000000 iterations - value = 3.14159265348480332
Time = 32872.7580 ms
```

20. 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 **SSED** – obliczenia z użyciem danych **double**, ale zrównoleglenie operacji pozwala na znaczące skrócenie czasu obliczeń w porównaniu do metody **CD** i **x87D**.
21. Testujemy inne liczby iteracji (np. $10^3$, $10^4$, $10^5$, $10^6$, $10^7$, $10^8$, $10^9$, $10^{10}$, $10^{11}$ – uwaga: czas obliczeń dla $10^{11}$ będzie ~100 razy dłuższy niż dla $10^9$) i rejestrujemy czasy działania funkcji **fun_cd**, **fun_ssed**, **fun_x87ld** i **fun_x87d** oraz uzyskaną dokładność (liczbę cyfr znaczących).