Politechnika Wrocławska

Wstęp do wysokowydajnych komputerów

Laboratorium nr. 3

Wykonawca:	
Imię i Nazwisko, nr indeksu, wydział	**Katarzyna Idasz** Wydział Informatyki i Telekomunikacji (W4N)
Termin zajęć: dzień tygodnia, godzina	Wtorek (11:15 – 13:00)
Numer grupy	1
Data wykonania ćwiczenia	30.04.2024 r.
Data oddania sprawozdania	13.05.2024 r.
Ocena końcowa	

Spis treści

1. Wstęp

1.1. Cel zadania

Należało napisać kod w C, który wywoła funkcję asemblerową z poprzedniego laboratorium oraz zmierzy czas jej wykonywania z wykorzystaniem rdtsc. Kod w asemblerze należało zmodyfikować tak, aby był zgodny z ABI.

1.2. Rdtsc

Rdtsc (Read Time Stamp Counter) to instrukcja procesora. Zwraca 64-bitową liczbę, reprezentującą liczbę cykli zegara procesora od momentu uruchomienia systemu, która zapisana jest w rejestrach edx:eax.

```
.global readTime
.type readTime, @function

readTime:
    push %ebx
    xor %eax, %eax
    cpuid
    rdtsc
    pop %ebx
    ret
```

Kod 1. time.s

1.3. Kod w C

Korzystając z rdtsc, jeżeli odczyta się liczbę cykli przed wywołaniem funkcji, a następnie po jej zakończeniu, to można odczytać ile cykli zajęło wykonanie danej funkcji przez odjęcie tych wartości od siebie.

```
#include <stdio.h>
extern long long int readTime();
extern void div();

int main() {

    long long int start = readTime();
    div();
    long long int end = readTime();
    long long int time = end - start;

    printf("%lld ticks\n", time);

    return 0;
}
```

Kod 2. kod.c

1.4. Dzielenie przez wielokrotne odejmowanie

Algorytm dzielenia to algorytm, który mając dane dwie liczby całkowite N i D (odpowiednio licznik i mianownik), oblicza ich iloraz i/lub resztę, wynik dzielenia euklidesowego.[1]

```
R := N
Q := 0
while R ≥ D do
  R := R − D
  Q := Q + 1
end
return (Q,R)
```

Kod 3. Pseudokod algorytmu dzielenia przez wielokrotne odejmowanie[2]

Aby kod z poprzednich laboratoriów był zgodny z ABI, zastosowano następujące modyfikacje:

- Przed *_start* dodano *div: push %ebx*
- Zmieniono etykietę *_start* na *_begin*
- Zmieniono .global _start na .global div
- Zakomentowano cały fragment kodu odpowiadający za wypisanie wyniku (etykieta *write*)
    - Zmodyfikowano *end* z:
      mov $EXIT_NR , %eax
      mov $EXIT_CODE_SUCCESS, %ebx
      int $0x80
      na:
      pop %ebx
      ret

Wszystkie zmiany zostały podkreślone w kodzie (4).

1.4.1. Omówienie kodu

- W *reset* out (czyli iloraz) jest resetowany poprzez ustawienie 0 na każdym bajcie liczby.
- W *dalej* wczytywane są dwie liczby oraz sprawdzane jest, czy nie skończyły się dane.
- *mainloop* odpowiada za główny algorytm.
- *compare* sprawdza warunek wyjścia z pętli, czyli czy reszta z dzielenia jest mniejsza niż dzielnik.
- *next* ustawia zmienne przed wejściem do pętli.
- *loop* to pętla, w której odejmowane są dwie liczby, bajt po bajcie.

[1] Division algorithm - Wikipedia
[2] Division algorithm - Division by repeated subtraction - Wikipedia

- *dodajjeden* - dodanie 1 do ilorazu - ustawia zmienne przed wejściem do pętli.
- *dodaj* to pętla, w której dodawana jest jedynka aż nie skończy się przeniesienie.
- *write* - wypisanie wyniku (iloraz i reszta).
- *end* - koniec programu.

```
EXIT_NR  = 1
READ_NR  = 3
WRITE_NR = 4

STDOUT = 1
STDIN = 0
EXIT_CODE_SUCCESS = 0

buff_len = 1
data_len = 32

.bss
    buff : .space buff_len
    first : .space data_len
    second : .space data_len
    out : .space data_len

.text

.global _div
.type _div, @function

div:
    push %ebx
_begin:
    mov $0, %eax
reset:
    cmp $data_len, %eax
    jz dalej
    movb $0, out(%eax)
    inc %eax
    jmp reset


dalej:
    mov $READ_NR,   %eax
    mov $STDIN,     %ebx
    mov $first,     %ecx
```

```
    mov $data_len,  %edx
    int $0x80

    cmp $data_len, %eax
    jl end

    mov $READ_NR,   %eax
    mov $STDIN,     %ebx
    mov $second,    %ecx
    mov $data_len,  %edx
    int $0x80

    mov $-1, %edi # Q; R = first; D = second

mainloop:
    mov $0, %eax
compare:
    cmp $data_len, %eax # Liczby są równe
    jz next

    movb first(%eax), %bl
    movb second(%eax), %cl
    inc %eax
    cmp %bl, %cl
    ja write

    cmp %bl, %cl
    jz compare


next:
    mov $data_len, %esi  # index
    mov $0, %cl          # przeniesienie


loop:
    dec %esi

    cmp $0, %esi
    jl dodajjeden

    movb first(%esi), %al     # Kopiuj bajt pierwszej liczby
    subb second(%esi), %al    # Odejmij bajt drugiej liczby
    setc %bl                  # Ustaw przeniesienie z odej liczb
    subb %cl, %al             # Odejmij poprzednie przeniesienie
    movb %al, first(%esi)     # Kopiuj bajt do wyniku
```

```
    setc %cl                    # Ustaw nowe przeniesienie z odej poprzedniego przeniesienia
    add %bl, %cl                # Dodaj przeniesienia -> do %cl

    jmp loop


dodajjeden:
    mov $data_len, %esi  # index
    mov $0, %cl          # przeniesienie

dodaj:
    dec %esi

    cmp $0, %esi
    jl mainloop

    mov out(%esi), %al
    add $1, %al
    movb %al, out(%esi)    # Kopiuj bajt do wyniku
    setc %cl
    cmp $0, %cl
    jnz dodaj
    jmp mainloop

write:
    #mov $WRITE_NR,   %eax
    #mov $STDOUT,     %ebx
    #mov $out,        %ecx
    #mov $data_len,   %edx
    #int $0x80

    #mov $WRITE_NR,   %eax
    #mov $STDOUT,     %ebx
    #mov $first,      %ecx
    #mov $data_len,   %edx
    #int $0x80

    jmp _begin

end:
    pop %ebx
    ret
```

Kod 4. lab4i5.s

2. Pomiary

Pomiary cykli przed optymalizacją oraz *perf* wykonano na laboratoriach. Ze względu na to, że pomiary należało wykonać ponownie po optymalizacji kodu oraz porównać, powtórzono je również na komputerze prywatnym.

2.1. Kompilacja

Do skompilowania wykorzystano:

$ as -g -32 time.s -o time.o

$ as -g -32 lab4i5.s -o lab4i5.o

$ gcc -g -m32 kod.c time.o lab4i5.o -o program

2.2. Procesor

Aby otrzymać informacje na temat procesora użyto *$ lscpu*.

2.2.1. Komputer laboratoryjny:

Architektura:	x86_64
Tryb(y) pracy CPU:	32-bit, 64-bit
Kolejność bajtów:	Little Endian
CPU:	4
Lista aktywnych CPU:	0-3
Wątków na rdzeń:	2
Rdzeni na gniazdo:	2
Gniazd:	1
Węzłów NUMA:	1
ID producenta:	GenuineIntel
Rodzina CPU:	6
Model:	60
Nazwa modelu:	Intel(R) Core(TM) i3-4160 CPU @ 3.60GHz
Wersja:	3
CPU MHz:	2032.851
CPU max MHz:	3600,0000
CPU min MHz:	800,0000
BogoMIPS:	7200.26
Wirtualizacja:	VT-x
Cache L1d:	32K
Cache L1i:	32K
Cache L2:	256K
Cache L3:	3072K

Procesory węzła NUMA 0: 0-3

Flagi: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 movbe popcnt tsc_deadline_timer aes

xsave avx f16c rdrand lahf_lm abm cpuid_fault invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt dtherm arat pln pts md_clear flush_l1d

2.2.2. Komputer prywatny:

Architektura:	x86_64
Tryb(y) pracy CPU:	32-bit, 64-bit
Address sizes:	44 bits physical, 48 bits virtual
Kolejność bajtów:	Little Endian
CPU:	12
Lista aktywnych CPU:	0-11
ID producenta:	AuthenticAMD
Nazwa modelu:	AMD Ryzen 5 4600H with Radeon Graphics
Rodzina CPU:	23
Model:	96
Wątków na rdzeń:	2
Rdzeni na gniazdo:	6
Gniazd:	1
Wersja:	1
Frequency boost:	enabled
CPU max MHz:	3000,0000
CPU min MHz:	1400,0000
BogoMIPS:	5988.57
Flagi:	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge m ca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep _good nopl nonstop_tsc cpuid extd_apicid aperfmperf ra pl pni pclmulqdq monitor ssse3 fma cx16 sse4_1 sse4_2 movbe popcnt aes xsave avx f16c rdrand lahf_lm cmp_leg acy svm extapic cr8_legacy abm sse4a misalignsse 3dnow prefetch osvw ibs skinit wdt tce topoext perfctr_core perfctr_nb bpext perfctr_llc mwaitx cpb cat_l3 cdp_l3 hw_pstate ssbd mba ibrs ibpb stibp vmmcall fsgsbase bm i1 avx2 smep bmi2 cqm rdt_a rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 cqm_llc cqm_occup _llc cqm_mbm_total cqm_mbm_local clzero irperf xsaveer ptr rdpru wbnoinvd cppc arat npt lbrv svm_lock nrip_sa ve tsc_scale vmcb_clean flushbyasid decodeassists paus efilter pfthreshold avic v_vmsave_vmload vgif v_spec_c trl umip rdpid overflow_recov succor smca sev sev_es

2.3. Perf

2.3.1. Komputer laboratoryjny:

2.3.1.1. Perf stat

```
ubuntu@xwdwk:~/Pulpit/temp/lab6i7$ perf stat ./program < dane/din32
340638870 ticks

 Performance counter stats for './program':

      93,299047   task-clock (msec)         #    0,974 CPUs utilized
            45    context-switches          #    0,482 K/sec
             0    cpu-migrations            #    0,000 K/sec
            40    page-faults               #    0,429 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses


      0,095784562 seconds time elapsed
```

Obraz 1. Perf stat dla komputera laboratoryjnego

2.3.1.2. Perf record & report

```
Samples: 38  of event 'cpu-clock', Event count (approx.): 9500000

Overhead  Command  Shared Object     Symbol

  52,63%  program  program           [.] loop
  18,42%  program  [vboxguest]       [k] VbglR0GRPerform
  18,42%  program  program           [.] compare
   2,63%  program  [kernel.kallsyms]  [k] VbglR0GRPerform
   2,63%  program  [kernel.kallsyms]  [k] __add_to_page_cache_locked
   2,63%  program  [kernel.kallsyms]  [k] __lock_text_start
   2,63%  program  program           [.] dodaj
```

Obraz 2. Perf record & report dla komputera laboratoryjnego

2.3.2. Komputer prywatny:

2.3.2.1. Perf stat

```
root@kass:/home/kasia/Pulpit/WdWK/lab6i7/cos# perf stat ./program < din32
255844530 ticks

 Performance counter stats for './program':

            88,20 msec task-clock                #    0,919 CPUs utilized
               16      context-switches          #  181,406 /sec
                0      cpu-migrations            #    0,000 /sec
               55      page-faults               #  623,582 /sec
      348 427 860      cycles                    #    3,950 GHz                      (84,78%)
        1 116 753      stalled-cycles-frontend   #    0,32% frontend cycles idle     (82,83%)
      165 867 677      stalled-cycles-backend    #   47,60% backend cycles idle      (81,86%)
    1 069 911 117      instructions              #    3,07  insn per cycle
                                                 #    0,16  stalled cycles per insn  (81,86%)
      241 010 775      branches                  #    2,733 G/sec                    (82,99%)
          100 261      branch-misses             #    0,04% of all branches          (85,69%)

      0,096002355 seconds time elapsed

      0,081788000 seconds user
      0,007789000 seconds sys
```

Obraz 3. Perf stat dla komputera prywatnego

2.3.2.2. Perf record & report

```
Samples: 428  of event 'cycles:P', Event count (approx.): 340653745
Overhead  Command  Shared Object      Symbol
  72,38%  program  program            [.] loop
  24,22%  program  program            [.] compare
   1,74%  program  program            [.] dodaj
   0,58%  program  program            [.] dodajjeden
   0,49%  program  program            [.] next
   0,29%  program  program            [.] mainloop
   0,15%  program  [kernel.kallsyms]  [k] copyout
   0,14%  program  ld-linux.so.2      [.] 0x0000000000024c94
   0,01%  program  [kernel.kallsyms]  [k] change_pte_range
   0,00%  perf-ex  [kernel.kallsyms]  [k] arch_perf_update_userpage
   0,00%  perf-ex  [kernel.kallsyms]  [k] perf_ibs_handle_irq
   0,00%  perf-ex  [kernel.kallsyms]  [k] native_write_msr
```

Obraz 4. Perf record & report dla komputera laboratoryjnego

2.3.3. Omówienie wyników

Wyniki perf record na obrazach (2) oraz (4) sygnalizują, że *loop* oraz *compare* mogą być obszarami, które warto rozpatrzyć przy optymalizacji kodu.

loop odpowiada za pętle, w której dodawane są dwie liczby, bajt po bajcie.

compare odpowiada za porównanie dwóch liczb - sprawdza warunek wyjścia z pętli, również bajt po bajcie.

3. Optymalizacja

3.1. Porównywanie liczb

Zamiast osobno porównywać liczby bajt po bajcie po każdym odjęciu dzielnika od reszty i sprawdzeniu czy reszta jest mniejsza od dzielnika, to po ich odjęciu jest sprawdzane, czy występuje przeniesienie. Jeśli tak, to znaczy, że wynik (reszta) jest ujemna, więc przed wypisaniem należy skorygować resztę poprzez dodanie do niej dzielnika w analogiczny sposób jak odejmowanie - *result*. W ten sposób pozbyto się się całego *compare*.

3.2. Resetowanie ilorazu

Zamiast ustawiać 0 na każdym bajcie ilorazu, skorzystano z rep stosb, który wypełnia obszar pamięci zerami:

- %edi - miejsce w pamięci, od którego chcemy zacząć wypełnianie
- %eax - a właściwie w %al - bajt który chcemy kopiować do pamięci
- %ecx - liczba bajtów, które mają być nadpisane

3.3. Zoptymalizowany kod

Reszta kodu pozostała bez zmian. Zmiany względem kodu (4) zostały podkreślone. Poprawność kodu została zweryfikowana na przykładowych danych.

```
EXIT_NR  = 1
READ_NR  = 3
WRITE_NR = 4

STDOUT = 1
STDIN = 0
EXIT_CODE_SUCCESS = 0

data_len = 32

.bss
    first : .space data_len
    second : .space data_len
    out : .space data_len

.text

.global div
.type div, @function

div:
    push %ebx
_begin:
    # Reset out - ustawienie na 0
    mov $out, %edi
```

```
    mov $0, %eax
    mov $data_len, %ecx
    rep stosb

    mov $READ_NR,   %eax
    mov $STDIN,     %ebx
    mov $first,     %ecx
    mov $data_len,  %edx
    int $0x80

    cmp $data_len, %eax
    jl end

    mov $READ_NR,   %eax
    mov $STDIN,     %ebx
    mov $second,    %ecx
    mov $data_len,  %edx
    int $0x80

    mov $0, %cl    # przeniesienie

mainloop:
    mov $data_len, %esi    # index


loop:
    dec %esi

    cmp $0, %esi
    jl dodajjeden

    movb first(%esi), %al     # Kopiuj bajt pierwszej liczby
    subb second(%esi), %al    # Odejmij bajt drugiej liczby
    setc %bl                  # Ustaw przeniesienie z odej liczb
    subb %cl, %al             # Odejmij poprzednie przeniesienie
    movb %al, first(%esi)     # Kopiuj bajt do wyniku
    setc %cl                  # Ustaw nowe przeniesienie z odej poprzedniego przeniesienia
    add %bl, %cl              # Dodaj przeniesienia -> do %cl

    jmp loop


dodajjeden:
    cmp $0, %cl
    jnz result     # Jesli przeniesienie > 0 to liczba jest ujemna wiec koniec algorytmu
```

```
    mov $data_len, %esi    # index

dodaj:
    dec %esi

    cmp $0, %esi
    jl mainloop

    mov out(%esi), %al
    add $1, %al
    movb %al, out(%esi)    # Kopiuj bajt do wyniku
    setc %cl
    cmp $0, %cl
    jnz dodaj
    jmp mainloop

result:
    # Korekcja wyniku
    mov $data_len, %esi       # index
    mov $0, %cl               # przeniesienie

    loop2:
    dec %esi

    cmp $0, %esi
    jl write

    movb first(%esi), %al     # Kopiuj bajt pierwszej liczby
    add second(%esi), %al     # Dodaj bajt drugiej liczby
    setc %bl                  # Ustaw przeniesienie z dodawania liczb
    add %cl, %al              # Dodaj poprzednie przeniesienie
    movb %al, first(%esi)     # Kopiuj bajt do wyniku
    setc %cl                  # Ustaw nowe przeniesienie z dod poprzedniego przeniesienia
    add %bl, %cl              # Dodaj przeniesienia -> do %cl

    jmp loop2

write:
    #mov $WRITE_NR,     %eax
    #mov $STDOUT,       %ebx
    #mov $out,          %ecx
    #mov $data_len,     %edx
    #int $0x80

    #mov $WRITE_NR,     %eax
    #mov $STDOUT,       %ebx
```

```
    #mov $first,        %ecx
    #mov $data_len,     %edx
    #int $0x80

    jmp _begin

end:
    pop %ebx
    ret
```

Kod 5. Zoptymalizowane lab4i5.s

3.4. Perf

```
root@kass:/home/kasia/Pulpit/WdWK/lab6i7/cos# perf stat ./program2 < din32
199179300 ticks

 Performance counter stats for './program2':

            67,25 msec task-clock                #    0,976 CPUs utilized
                0      context-switches          #    0,000 /sec
                0      cpu-migrations            #    0,000 /sec
               49      page-faults               #  728,600 /sec
      266 596 817      cycles                    #    3,964 GHz                      (82,16%)
          470 202      stalled-cycles-frontend   #    0,18% frontend cycles idle     (82,16%)
      119 703 509      stalled-cycles-backend    #   44,90% backend cycles idle      (82,16%)
      781 937 267      instructions              #    2,93  insn per cycle
                                                 #    0,15  stalled cycles per insn  (82,15%)
      146 699 406      branches                  #    2,181 G/sec                    (86,09%)
           30 115      branch-misses             #    0,02% of all branches          (85,29%)

      0,068930629 seconds time elapsed

      0,069100000 seconds user
      0,000000000 seconds sys
```

Obraz 5. Perf stat po optymalizacji

```
Samples: 499  of event 'cycles:P', Event count (approx.): 269489520
Overhead  Command   Shared Object       Symbol
  95,73%  program2  program2            [.] loop
   1,74%  program2  program2            [.] dodaj
   1,42%  program2  program2            [.] dodajjeden
   0,37%  program2  [kernel.kallsyms]   [k] clear_page_rep
   0,37%  program2  program2            [.] mainloop
   0,13%  program2  [kernel.kallsyms]   [k] vm_get_page_prot
   0,12%  program2  [kernel.kallsyms]   [k] mmap_region
   0,09%  program2  [kernel.kallsyms]   [k] memcg_account_kmem
   0,03%  program2  [kernel.kallsyms]   [k] obj_cgroup_charge
   0,01%  program2  [kernel.kallsyms]   [k] chacha_block_generic
   0,00%  perf-exe  [kernel.kallsyms]   [k] visit_groups_merge.constprop.0.isra.0
   0,00%  perf-exe  [kernel.kallsyms]   [k] perf_event_idx_default
   0,00%  perf-exe  [kernel.kallsyms]   [k] native_write_msr
```

Obraz 6. Perf record & report po optymalizacji

3.4.1. Omówienie wyników

Z obrazów (3) oraz (5):

- Czas wykonywania programu zmniejszył się z 88,20ms na 67,25ms -> około 24% poprawa.
- Ilość cykli zmniejszyła się z około 348 milionów na około 266 milionów -> około 23% poprawa.
- Ilość instrukcji zmniejszyła się z około 1 miliarda na około 782 miliony -> około 22% poprawa.

Z obrazów (4) oraz (6):

- Pozbyto się *compare*, natomiast teraz *loop* zajmuje najwięcej czasu ogólnie, jest to jednak nieuniknione ze względu na specyfikację zadania, w którym długie liczby są dodawane do siebie bajt po bajcie.

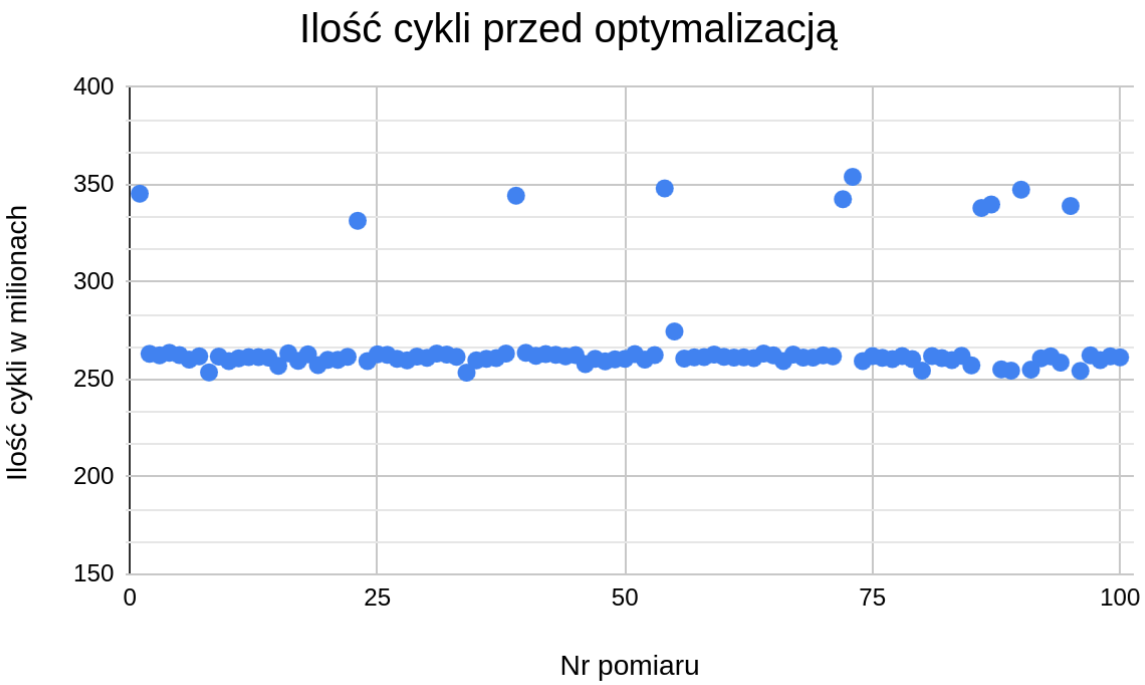
4. Rezultaty

Po kompilacji, skorzystano z *$ for i in {1..100}; do ./program < din32; done*, aby pobrać wyniki - ile cykli zajęło wykonanie dzielenia przez wielokrotne odejmowanie. Zarówno przed i po optymalizacji wykonano 100 pomiarów.

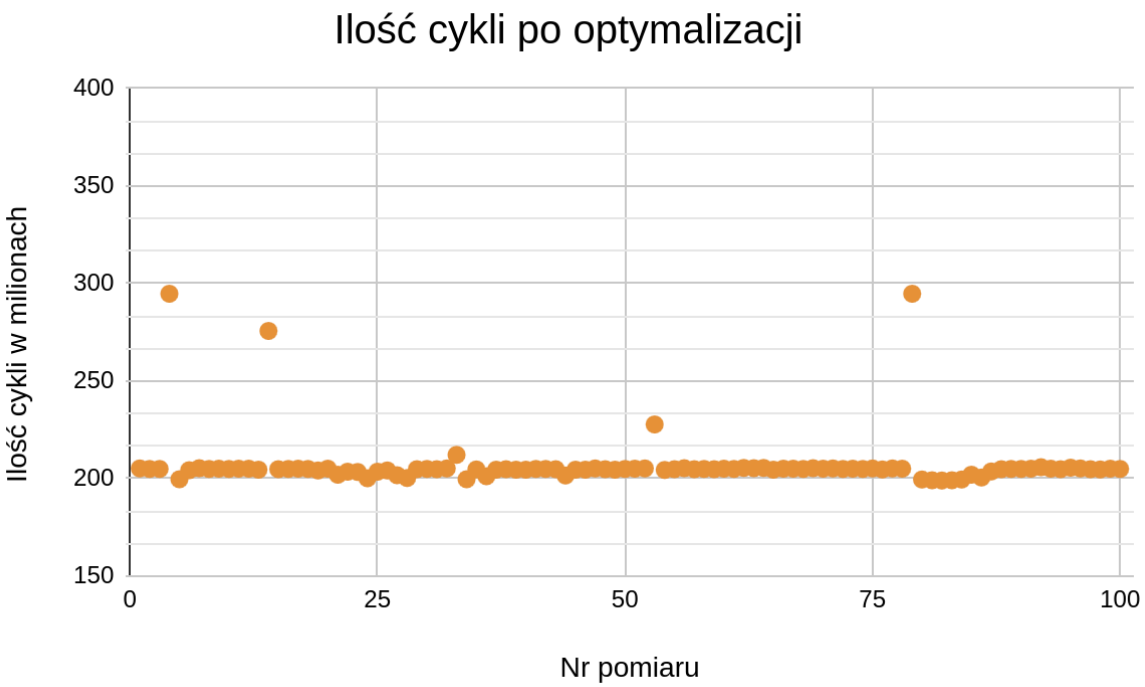
W tabeli (1) zakreślono na szaro pomiary, które nie są brane pod uwagę w analizie, ponieważ są tą błędy grube - rdtsc nie jest idealnym narzędziem. Najlepiej widać to na wykresach (1) i (2) - pomiary są oddalone od pozostałych. Po ich usunięciu wykres przypomina prostą.

Tab 1. Ilość cykli przed i po optymalizacji - komputer prywatny

Nr pomiaru			Ilość cykli przed optymalizacją			Ilość cykli po optymalizacji		
1	35	68	345388740	259813260	261238740	205177950	204666690	204951840
2	36	69	263149410	260678130	261269880	204910470	201287400	205311090
3	37	70	262370400	261004260	262475220	205015800	204616410	205039830
4	38	71	263669640	263324340	261903210	294815460	204881910	205278420
5	39	72	262504140	344398470	342578220	199707360	204617850	204959910
6	40	73	260303520	263668050	354013680	204333660	204612540	205031280
7	41	74	261856080	262120140	259517640	205339260	204927570	204979410
8	42	75	253699410	263014620	262037130	205003050	204946380	205265010
9	43	76	261753600	262690680	261091650	205097760	204868560	204677220
10	44	77	259488660	261888150	260523510	204938760	201608040	205225800
11	45	78	260899140	262532610	261998490	205138650	204541440	205076760
12	46	79	261566580	257921850	260529780	205144230	204638400	294709920
13	47	80	261571170	260638080	254566380	204597420	205259340	199511100
14	48	81	261330870	259366050	262016010	275652750	204795540	199110150
15	49	82	257064480	260425050	261045390	204826980	204623070	199032360
16	50	83	263478390	260597490	259937520	205018350	204960570	199224330
17	51	84	259650000	263016360	262213380	205043370	205104330	199618920
18	52	85	262958490	260217780	257333550	204945630	205268670	202006500
19	53	86	257406150	262521120	338032170	204238740	227830980	200636550
20	54	87	260166540	348071940	339886590	205085040	204437880	203689560
21	55	88	260145600	274619610	255183330	201951480	204882870	204896970
22	56	89	261655110	260697240	254517360	203506950	205329750	205001310
23	57	90	331432830	261389010	347438940	203379480	204859710	204914160
24	58	91	259442190	261560670	255034680	200175660	204938490	205136250
25	59	92	262943880	262850940	260903340	203487930	204835980	205915290
26	60	93	262720710	261636540	261945960	204231330	205084470	205106910
27	61	94	260642760	261265620	258686730	201773670	204963180	204829170
28	62	95	259791780	261354600	339157440	200301810	205523310	205611270
29	63	96	261763590	261073200	254470740	204782010	205366290	205166850
30	64	97	261098730	263298930	262452390	205009020	205539750	204887970
31	65	98	263349810	262381500	260015790	204861420	204607380	204768390
32	66	99	262780500	259467360	261909120	205163220	205147500	205144140
33	67	100	261608250	262796250	261402420	212150250	205133820	205003410
34			253481280			199726200		



Wyk. 1 Ilość cykli przed optymalizacją - komputer prywatny

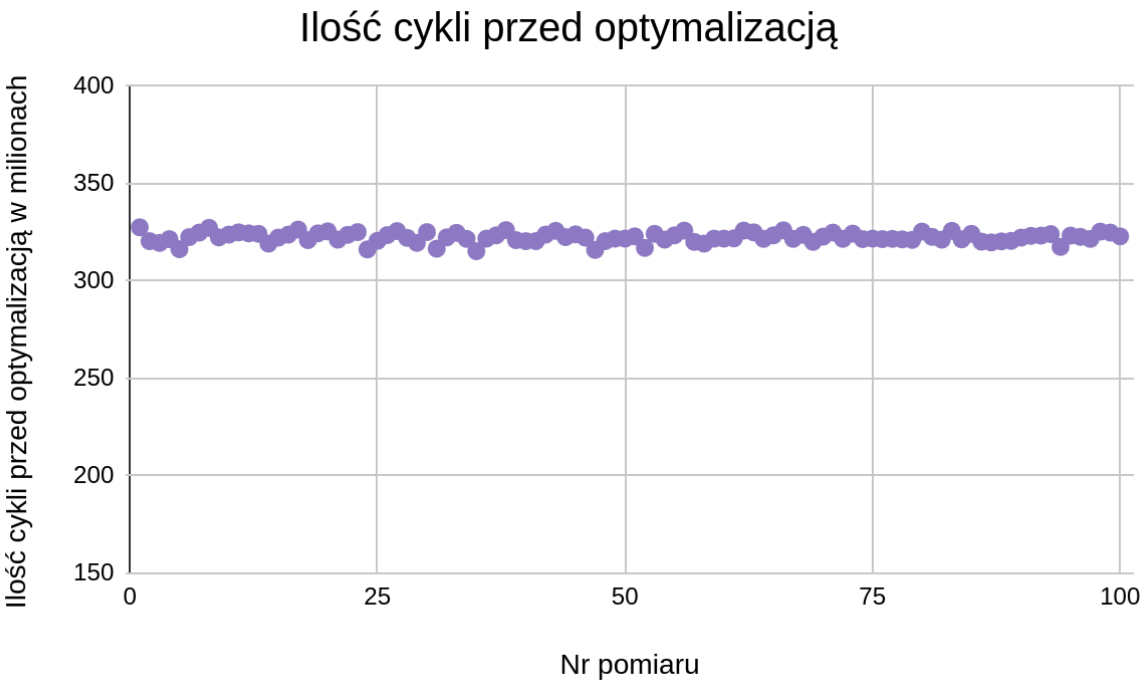


Wyk. 2 Ilość cykli po optymalizacji - komputer prywatny

Tab 2. Ilość cykli przed i po optymalizacji - komputer laboratoryjny

Nr pomiaru			Ilość cykli przed optymalizacją		
1	35	68	327580232	315359288	323720504
2	36	69	320526712	321816232	320195452
3	37	70	319582148	323528848	322852892
4	38	71	321574460	326170308	324988292
5	39	72	316433316	321013948	321721052
6	40	73	322578608	320546840	324444812
7	41	74	324890672	320603564	321575160
8	42	75	327398752	323855676	321786384
9	43	76	322482640	325885424	321632720
10	44	77	323843836	322579920	321715040
11	45	78	325066192	323978448	321444416
12	46	79	324587928	322321416	321160180
13	47	80	324262068	316050040	325454732
14	48	81	319316456	320487948	322713780
15	49	82	322290168	321844836	321315572
16	50	83	323858584	321822128	325781432
17	51	84	326451892	323050492	321441472
18	52	85	321061996	317044068	324324232
19	53	86	324550456	324227928	320342744
20	54	87	325565324	321317576	319923164
21	55	88	321262112	323505588	320365840
22	56	89	323757340	326008692	320612300
23	57	90	325188676	320139156	322292428
24	58	91	316239420	319245228	323218540
25	59	92	320718364	321866920	323346172
26	60	93	323624764	321836932	324180048
27	61	94	325677188	322004188	317559152
28	62	95	322168564	325980608	323330220
29	63	96	319589268	325084756	322742964
30	64	97	325149572	321725764	321665336
31	65	98	316648856	323469620	325441624
32	66	99	322423288	326031372	324978384
33	67	100	324795668	321721084	323018800
34			321752376		

W tabeli (2) nie znaleziono błędów grubych - brak odstających wartości, co widać na wykresie (3).



Wyk. 3 Ilość cykli przed optymalizacją - komputer laboratoryjny

Tab 3. Analiza statystyczna pomiarów - komputer prywatny

	Przed optymalizacją	Po optymalizacji
Średnia	260915107,67	204270272,19
Mediana	261342735,00	204920865,00
Odchylenie std.	2745869,88	1925923,25

Tab 4. Analiza statystyczna pomiarów - komputer laboratoryjny

	Przed optymalizacją
Średnia	322463085,72
Mediana	322372352,00
Odchylenie std.	2529536,59

4.1. Wnioski

- Wydajność średnia wzrosła (oraz średnia ilość cykli zmniejszyła się) o: $\frac{260915107,67-204270272,19}{260915107,67} \cdot 100\% \approx 21,7\%$
- Średnia i mediana są zbliżone do siebie - nie ma wartości skrajnych, które miałyby wpływ na średnią, czyli udało się je odrzucić.
- W obu przypadkach wartość odchylenia standardowego jest względnie niska w porównaniu do średniej (odchylenie std. to około 1% średniej). Oznacza to, że większość danych jest zbliżona do średniej, ale istnieje pewna zmienność.