Organizacja i architektura komputerów - projekt

Raport końcowy.

Implementacja biblioteki arytmetyki liczb stałoprzecinkowych dowolnej precyzji z wykorzystaniem wewnętrznej reprezentacji z obciążeniem.

16.05.2019

Prowadzący: Dr. Inż. Tadeusz Tomczak
Skład grupy: Wojciech Gąsiewicz 235086

Spis treści:

1. Cel projektu.

Celem projektu była implementacja biblioteki arytmetyki liczb stałoprzecinkowych dowolnej precyzji z wykorzystaniem wewnętrznej reprezentacji z obciążeniem. Podczas implementacji należało wykorzystać również wsparcie oferowane przez procesor.

2. Wykorzystane technologie.

Biblioteka została zaimplementowana w języku C z wykorzystaniem assemblera AT&T. Testy były przeprowadzane na komputerze wyposażonym w procesor i7-3720QM posiadający 32GB pamięci RAM. Wykorzystano system operacyjny Centos 7.

3. Struktura danych.

Do zrealizowania założeń systemu z obciążeniem zaimplementowana została poniższa struktura:

```
struct any_BIAS
{
unsigned char * digits;
unsigned long length;
};
```

Ciąg cyfr z których składa się liczba przechowywany jest w zmiennej digits. Zmienna length natomiast określa długość liczby z obciążeniem.

4. Funkcje zaimplementowane w bibliotece:

- **int Any_Add_BIAS(any_BIAS_p a, any_BIAS_p b);** - funkcja realizująca operację dodawania.
- **int Any_Sub_BIAS(any_BIAS_p a, any_BIAS_p b);** - funkcja realizująca operację odejmowania.
- **int Any_Mul_BIAS(any_BIAS_p a, any_BIAS_p b);** - funkcja realizująca operację mnożenia.
- **int Any_Div_BIAS(any_BIAS_p a, any_BIAS_p b);** - funkcja realizująca operację dzielenia.
- **int Any_FromString(any_BIAS_p a, unsigned char * b);** - funkcja realizująca operację wczytania liczby od użytkownika.
- **int Any_output(any_BIAS_p a);** - funkcja realizująca operację wyświetlania podanej w parametrze liczby.

5. Zaimplementowane moduły:

Biblioteka została zaimplementowana tak aby realizować podstawowe operacje matematyczne wykorzystując reprezentacje z obciążeniem. Wartość obciążenia zależy od długości liczby. Obliczamy ją ze wzoru:

$$n = \frac{10^k}{2} - 1$$

Pozwala to na zakodowanie liczby z zakresu od $-(\frac{10^n}{2} - 1)$ do $\frac{10^n}{2}$

5.1 Dodawanie.

Operacja dodawania opiera się na instrukcji **adc**. Umożliwia ona dodanie zawartości dwóch rejestrów uwzględniając przy tym flagę carry flag. Działanie wykonuje się według wzoru:

DEST ← DEST + SRC + CF [1].

Funkcja realizująca dodawanie int **any_Add_BIAS()** przyjmuje w parametrach dwie liczby w systemie z obciążeniem. Następnie cyfry obu liczb dodawane są do siebie z wykorzystaniem wyżej opisanej instrukcji adc. Po wykonanej operacji dodawania od wyniku odejmowane jest jedno obciążenie. Jeśli wystąpi konieczność zmiany obciążenia ze względu na przekroczenie zakresu generowane jest nowe, które pozwala na zakodowanie liczby o większej wartości. Jeżeli metoda wykona się poprawnie to zwracana jest wartość 0 w przeciwnym przypadku 1. Wartość wykonanej operacji zapisywana jest w pierwszym parametrze przekazywanym do funkcji.

5.2 Odejmowanie.

Operacja odejmowania opiera się na instrukcji **sbb**. Działanie tej instrukcji jest analogiczne do instrukcji adc. Odpowiada ona za odejmowanie wartości dwóch rejestrów uwzględniając przy tym flagę carry flag. Operacja wykonuje się według wzoru:

DEST ← DEST – (SRC + CF)[1].

Funkcja realizująca odejmowanie int **any_Sub_Bias()** przyjmuje w parametrach dwie liczby w systemie z obciążeniem. Cyfry obu liczb są następnie odejmowane za pomocą opisanej powyżej instrukcji sbb. Jeżeli metoda wykona się poprawnie to zwracana jest wartość 0 w przeciwnym przypadku 1. Wartość wykonanej operacji zapisywana jest w pierwszym parametrze przekazywanym do funkcji.

5.3 Mnożenie.

Funkcja realizująca mnożenie int **any_Mul_Bias()** przyjmuje w parametrach dwie liczby w systemie z obciążeniem. Operacja mnożenia jest realizowana przez obliczanie oraz sumowanie kolejnych iloczynów częściowych. Sposób ten pozwala ograniczyć ilość wykonywanych operacji dodawania, co przekłada się na krótszy czas wykonywania. Do jej realizacji wykorzystywana jest metoda dodawania opisana w punkcie 5.1.
Jeżeli metoda wykona się poprawnie to zwracana jest wartość 0 w przeciwnym przypadku 1. Wartość wykonanej operacji zapisywana jest w pierwszym parametrze przekazywanym do funkcji.

5.4 Dzielenie.

Funkcja realizująca dzielenie int **any_Div_Bias()** przyjmuje w parametrach dwie liczby w systemie z obciążeniem. Sposób wykonywania operacji dzielenia jest oparty na operacji odejmowania. Aby przyspieszyć proces dla dużych liczb dzielnik uzupełniany jest cyframi 0 z prawej strony tak, aby wyrównać jego długość do dzielnej. Następnie algorytm w pętli odejmuje od dzielnej wartość dzielnika. Przy każdej iteracji wartość wyniku jest zwiększana o 1 oraz jest mnożona przez 10 poprzez dopisanie do jego aktualnej wartości 0 z prawej

strony. Dzielenie kończy się, gdy aktualna długość dzielnika będzie mniejsza od jego początkowej długości. Jeżeli metoda wykona się poprawnie to zwracana jest wartość 0 w przeciwnym przypadku 1. Wartość wykonanej operacji zapisywana jest w pierwszym parametrze przekazywanym do funkcji.

6. Kompilacja biblioteki.

Do kompilacji programu oraz biblioteki wykorzystane zostało narzędzie **make**. Zawartość pliku makefile jest następująca:

```
default: libtest
clear:
	rm -f libtest
arthmeticlib: arithmeticlib.c calculation.s
	gcc –Wall –O1  -std=c99 arthmeticlib.c calculation.s -o arthmeticlib
libtest: libarthmeticlib.a libtest.c
	gcc –Wall –O1 -std=c99 -g libtest.c calculation.s -L. -larthmeticlib -lcunit -o libtest
libarthmeticlib.a: arthmeticlib.o calculation.s
	ar -rcsv libarthmeticlib.a calculation.s arthmeticlib.o
```

Opcja **-Wall** odpowiada za wyświetlanie wszystkich ostrzeżeń na temat kompilowanego kodu. Ułatwia ona wykrywanie oraz poprawę ewentualnych błędów. **-std=99** oznacza, że kod skompilowany będzie w standardzie języka C z 1999 roku.
Flaga **-O1**[2] oznacza pierwszy poziom optymalizacji kodu przez kompilator gcc. Przeklada sie na dluzszy czas kompilacji oraz szybsza prace programu.
Program ar służy min. do utworzenia archiwum. Opcja **r** odpowiada za umieszczenie plików w archiwum, **c** za utworzenie archiwum, **s** zapisuje plik obiektowy do archiwum lub uaktualnia go, jeśli już istnieje oraz **v** tworzy opis archiwum[3].

7. Profilowanie kodu.

W celu dynamicznej analizy kodu zastosowane zostało profilowanie, które pozwoliło na określenie „wąskich gardeł”.

Do profilowania zaimplementowanej biblioteki wykorzystany został program perf[5]. Jest to narzędzie dostępne w systemie linux służące do badania analizy wydajności. Może być wykorzystywane do profilowania zarówno jądra systemu operacyjnego jak i kodu użytkownika.

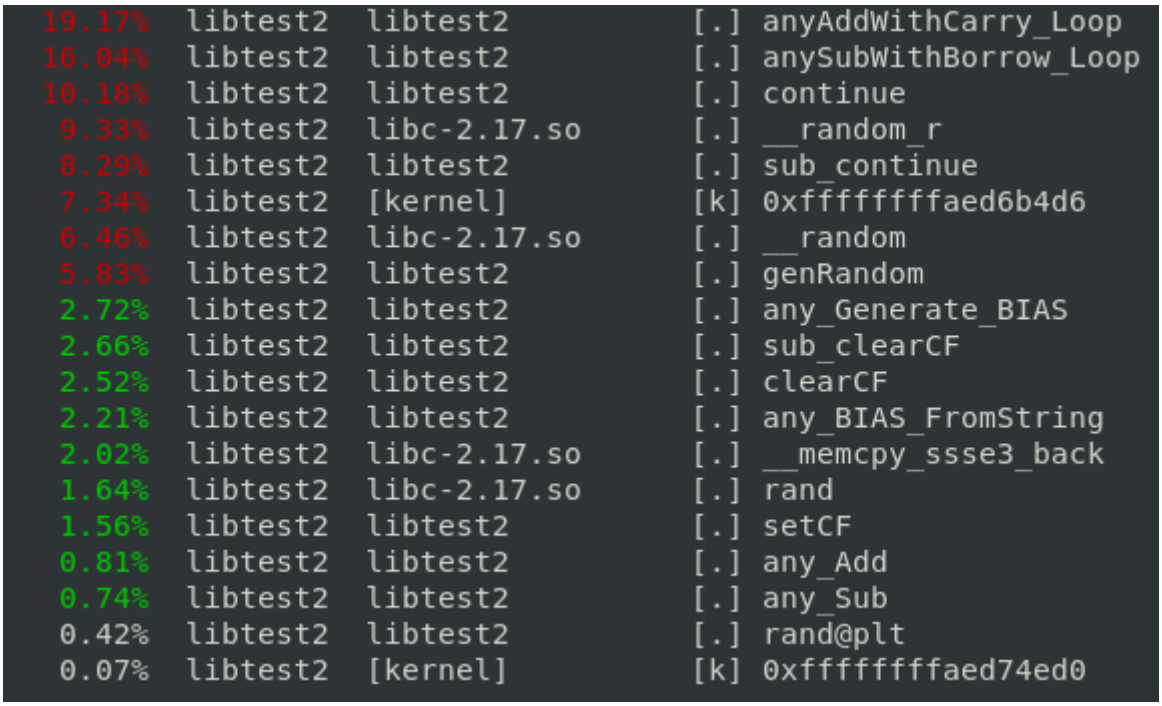
Instrukcja rozpoczynająca profilowanie:

```
perf record ./nazwa_programu
```

Do wyświetlenia wyników konieczne jest użycie instrukcji:

```
perf report
```

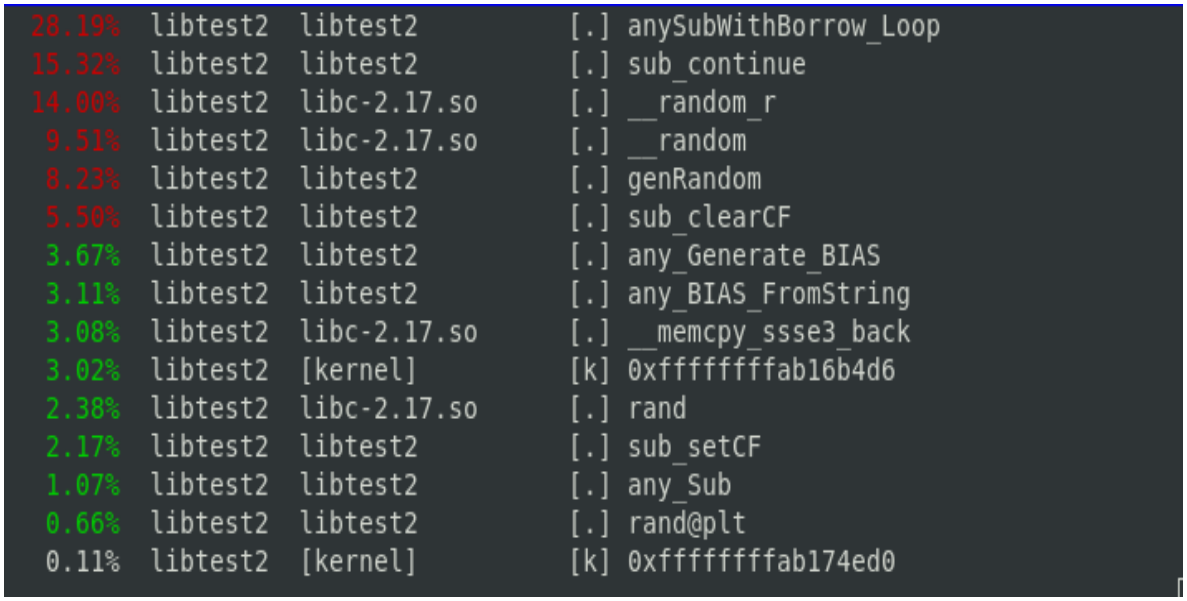
Poniżej przedstawiono zrzut ekranu zawierający przykładowy raport:



19.17%	libtest2	libtest2	[.]	anyAddWithCarry_Loop
16.04%	libtest2	libtest2	[.]	anySubWithBorrow_Loop
10.18%	libtest2	libtest2	[.]	continue
9.33%	libtest2	libc-2.17.so	[.]	__random_r
8.29%	libtest2	libtest2	[.]	sub_continue
7.34%	libtest2	[kernel]	[k]	0xffffffffaed6b4d6
6.46%	libtest2	libc-2.17.so	[.]	__random
5.83%	libtest2	libtest2	[.]	genRandom
2.72%	libtest2	libtest2	[.]	any_Generate_BIAS
2.66%	libtest2	libtest2	[.]	sub_clearCF
2.52%	libtest2	libtest2	[.]	clearCF
2.21%	libtest2	libtest2	[.]	any_BIAS_FromString
2.02%	libtest2	libc-2.17.so	[.]	__memcpy_ssse3_back
1.64%	libtest2	libc-2.17.so	[.]	rand
1.56%	libtest2	libtest2	[.]	setCF
0.81%	libtest2	libtest2	[.]	any_Add
0.74%	libtest2	libtest2	[.]	any_Sub
0.42%	libtest2	libtest2	[.]	rand@plt
0.07%	libtest2	[kernel]	[k]	0xffffffffaed74ed0

Rysunek 1 Perf - raport dodawanie.

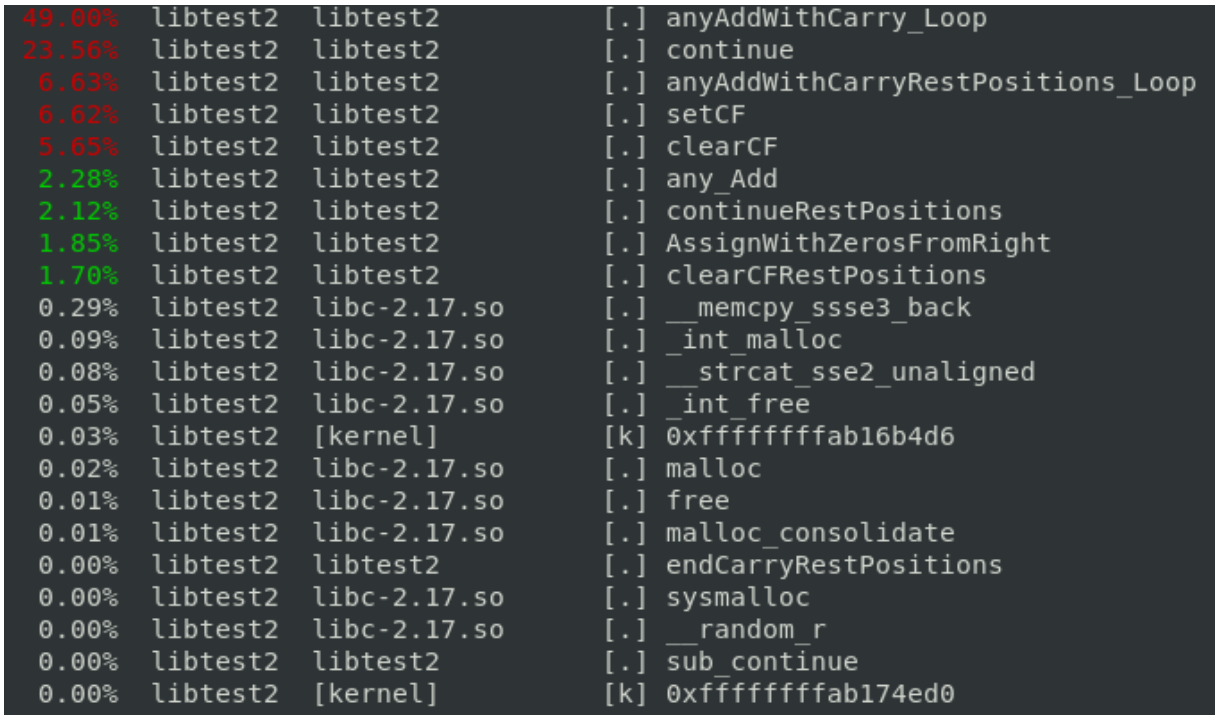
Powyższy raport został wygenerowany na przykładzie funkcji dodawania dwóch liczb z obciążeniem składających się z 100000000 cyfr każda. Jak widać w raporcie program najwięcej czasu spędził w funkcjach anyAddWithCarry_Loop oraz anySubWithBorrow_Loop. Są to funkcje assemblerowe realizujące odpowiednio dodawanie oraz odejmowania przy użyciu instrukcji adc oraz sbb.



28.19%	libtest2	libtest2	[.]	anySubWithBorrow_Loop
15.32%	libtest2	libtest2	[.]	sub_continue
14.00%	libtest2	libc-2.17.so	[.]	__random_r
9.51%	libtest2	libc-2.17.so	[.]	__random
8.23%	libtest2	libtest2	[.]	genRandom
5.50%	libtest2	libtest2	[.]	sub_clearCF
3.67%	libtest2	libtest2	[.]	any_Generate_BIAS
3.11%	libtest2	libtest2	[.]	any_BIAS_FromString
3.08%	libtest2	libc-2.17.so	[.]	__memcpy_ssse3_back
3.02%	libtest2	[kernel]	[k]	0xffffffffab16b4d6
2.38%	libtest2	libc-2.17.so	[.]	rand
2.17%	libtest2	libtest2	[.]	sub_setCF
1.07%	libtest2	libtest2	[.]	any_Sub
0.66%	libtest2	libtest2	[.]	rand@plt
0.11%	libtest2	[kernel]	[k]	0xffffffffab174ed0

Rysunek 2 Perf - raport odejmowanie.

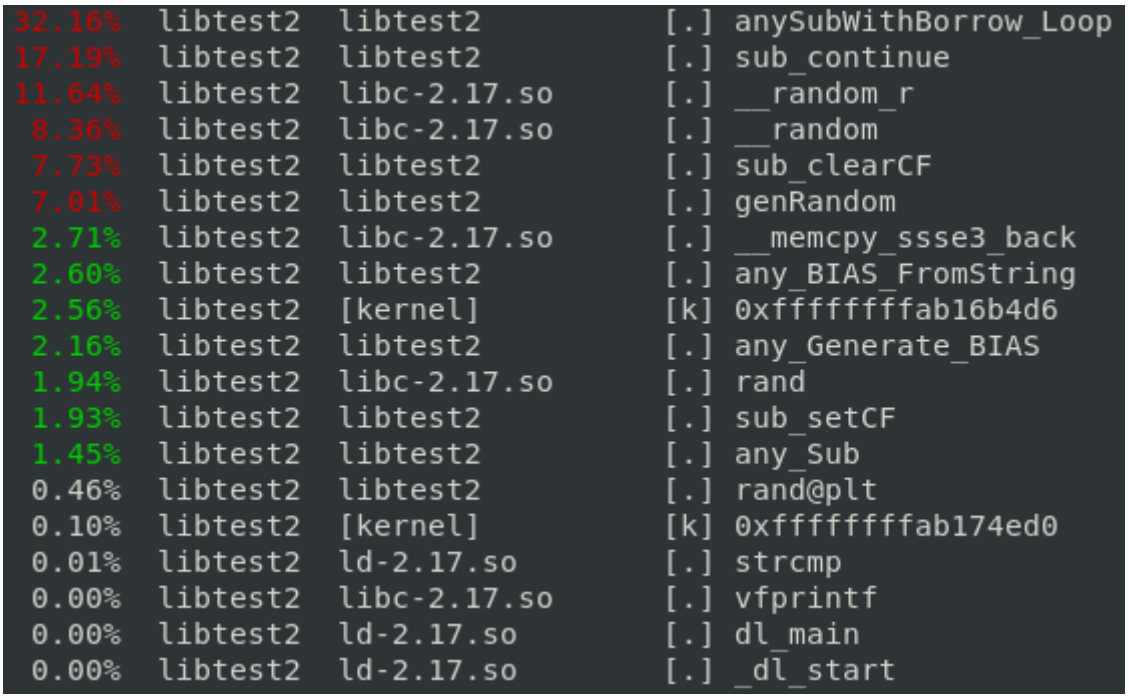
Dla operacji odejmowania uzyskujemy podobne wyniki jak w przypadku dodawania.



```
49.00%  libtest2  libtest2          [.] anyAddWithCarry_Loop
23.56%  libtest2  libtest2          [.] continue
 6.63%  libtest2  libtest2          [.] anyAddWithCarryRestPositions_Loop
 6.62%  libtest2  libtest2          [.] setCF
 5.65%  libtest2  libtest2          [.] clearCF
 2.28%  libtest2  libtest2          [.] any_Add
 2.12%  libtest2  libtest2          [.] continueRestPositions
 1.85%  libtest2  libtest2          [.] AssignWithZerosFromRight
 1.70%  libtest2  libtest2          [.] clearCFRestPositions
 0.29%  libtest2  libc-2.17.so      [.] __memcpy_ssse3_back
 0.09%  libtest2  libc-2.17.so      [.] _int_malloc
 0.08%  libtest2  libc-2.17.so      [.] __strcat_sse2_unaligned
 0.05%  libtest2  libc-2.17.so      [.] _int_free
 0.03%  libtest2  [kernel]          [k] 0xffffffffab16b4d6
 0.02%  libtest2  libc-2.17.so      [.] malloc
 0.01%  libtest2  libc-2.17.so      [.] free
 0.01%  libtest2  libc-2.17.so      [.] malloc_consolidate
 0.00%  libtest2  libtest2          [.] endCarryRestPositions
 0.00%  libtest2  libc-2.17.so      [.] sysmalloc
 0.00%  libtest2  libc-2.17.so      [.] __random_r
 0.00%  libtest2  libtest2          [.] sub_continue
 0.00%  libtest2  [kernel]          [k] 0xffffffffab174ed0
```

Rysunek 3 Perf - raport mnożenie.

Operacja mnożenia prawie połowę czasu wykonywania programu spędza w pętli realizującej dodawanie. Jest to zgodne z oczekiwaniem ponieważ zastosowany algorytm sum częściowych polega na wielokrotnym dodawaniu.



```
32.16%  libtest2  libtest2          [.] anySubWithBorrow_Loop
17.19%  libtest2  libtest2          [.] sub_continue
11.64%  libtest2  libc-2.17.so      [.] __random_r
 8.36%  libtest2  libc-2.17.so      [.] __random
 7.73%  libtest2  libtest2          [.] sub_clearCF
 7.01%  libtest2  libtest2          [.] genRandom
 2.71%  libtest2  libc-2.17.so      [.] __memcpy_ssse3_back
 2.60%  libtest2  libtest2          [.] any_BIAS_FromString
 2.56%  libtest2  [kernel]          [k] 0xffffffffab16b4d6
 2.16%  libtest2  libtest2          [.] any_Generate_BIAS
 1.94%  libtest2  libc-2.17.so      [.] rand
 1.93%  libtest2  libtest2          [.] sub_setCF
 1.45%  libtest2  libtest2          [.] any_Sub
 0.46%  libtest2  libtest2          [.] rand@plt
 0.10%  libtest2  [kernel]          [k] 0xffffffffab174ed0
 0.01%  libtest2  ld-2.17.so        [.] strcmp
 0.00%  libtest2  libc-2.17.so      [.] vfprintf
 0.00%  libtest2  ld-2.17.so        [.] dl_main
 0.00%  libtest2  ld-2.17.so        [.] _dl_start
```

Rysunek 4 Perf - raport dzielenie.

Operacja dzielenia najwięcej czasu spędza w funkcji realizującej odejmowanie. Dzieje się tak ze względu na zastosowanie algorytmu „dzielenia pisemnego”, które polega na wielokrotnym odejmowaniu dzielnika.

Sprawdzenie programu pod względem braku wycieków pamięci zostało zrealizowane przy pomocy programu **valgrind**[6]. Instrukcja pozwalająca zrealizować taka operacja wygląda następująco:

```
valgrind --leak-check=yes ./nazwa_programu
```

Poniżej przedstawiony został przykładowy wynik powyższej instrukcji:

```
==21422== HEAP SUMMARY:
==21422==     in use at exit: 0 bytes in 0 blocks
==21422==   total heap usage: 852 allocs, 852 frees, 6,354 bytes allocated
==21422==
==21422== All heap blocks were freed -- no leaks are possible
==21422==
==21422== For counts of detected and suppressed errors, rerun with: -v
==21422== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Rysunek 5 Valgrind - wynik badania wycieków pamięci.

8. Wyniki testów wydajnościowych.

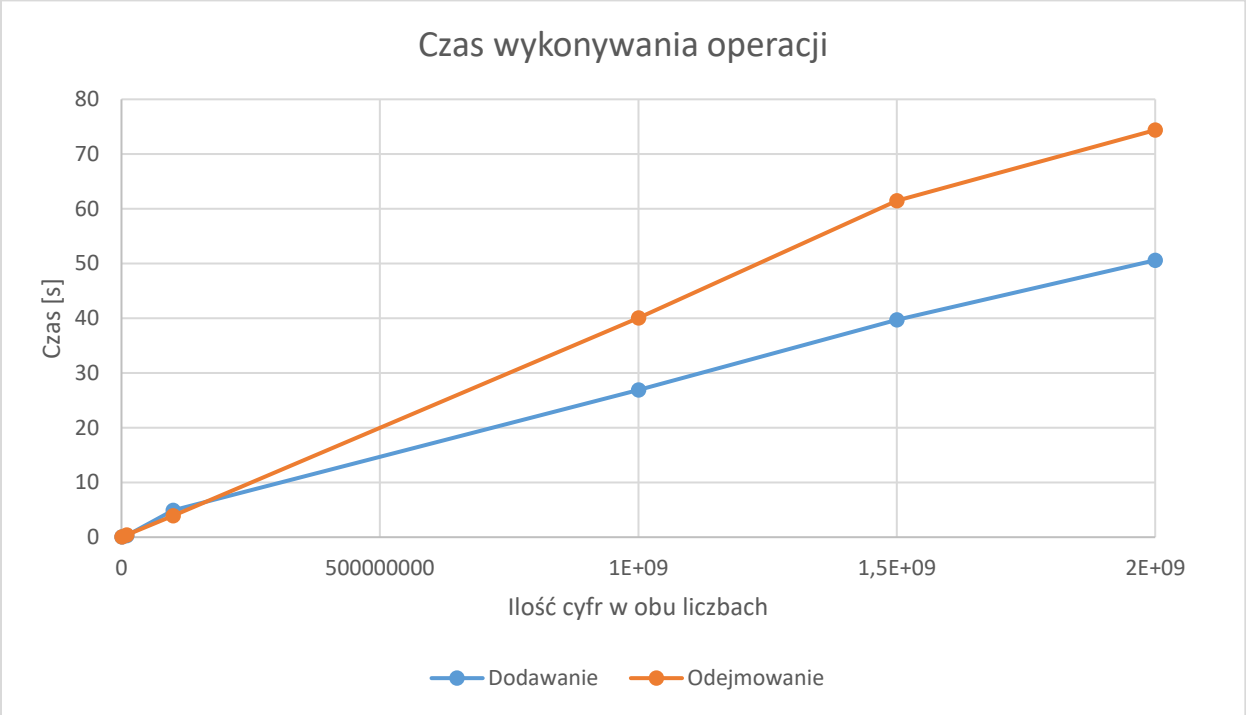
Testy wydajnościowe zostały przeprowadzone dla dwóch losowych liczb posiadających identyczną ilość cyfr. Zakres ilości cyfr został dobrany od 1000000 do 2000000000. Czas mierzony był w sekundach przy użyciu funkcji clock z biblioteki „time.h”. Ilość używanej przez program pamięci ram była sprawdzana za pomocą programu top.

Ilość cyfr	Czas operacji [s]	Użycie pamięci RAM [GB]
1000000	0,05	
10000000	0,29	
100000000	4,91	0,652
1000000000	26,88	6,5
1500000000	39,72	9,8
2000000000	50,6	11,2

Tabela 1 Dodawanie bez optymalizacji

Ilość cyfr	Czas operacji [s]	Użycie pamięci RAM
1000000	0,07	
10000000	0,4	
100000000	3,92	0,696
1000000000	40,06	8
1500000000	61,47	10,1
2000000000	74,38	14,3

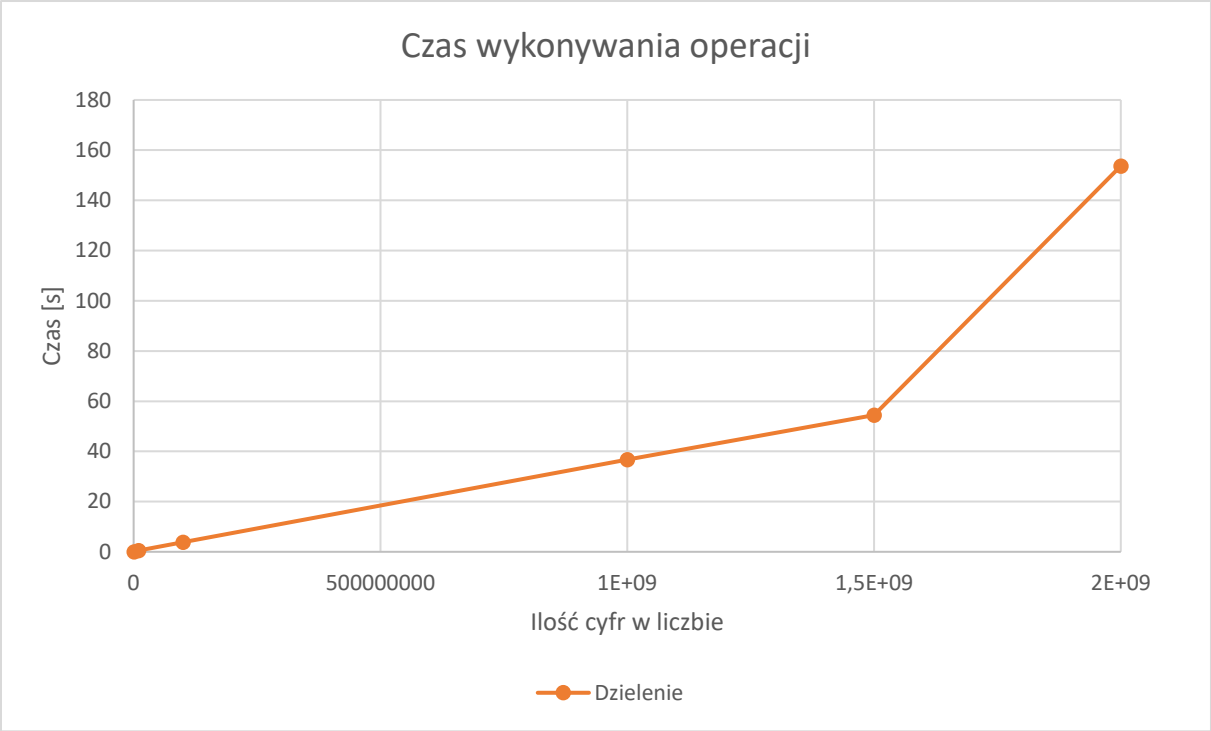
Tabela 2 Odejmowanie bez optymalizacji.



Wykres 1 Czas wykonywania dodawania oraz odejmowania bez optymalizacji.

Ilość cyfr	Czas operacji [s]	Użycie pamięci RAM
1000000	0,06	
10000000	0,5	
100000000	3,79	0,97
1000000000	36,76	6,5
1500000000	54,45	9,8
2000000000	153,7	13,2

Tabela 3 Dzielenie bez optymalizacji



Wykres 2 Czas wykonywania dzielenia bez optymalizacji.

Ilość cyfr	Czas operacji [s]
3000	0,77
6250	3,35
12500	13,31
25000	53,25
50000	209,54

Tabela 4 Mnożenie bez optymalizacji.

Wykres 3 Czas wykonywania mnożenia bez optymalizacji.

9. Wyniki testów wydajnościowych z użyciem optymalizacji kompilatora gcc.

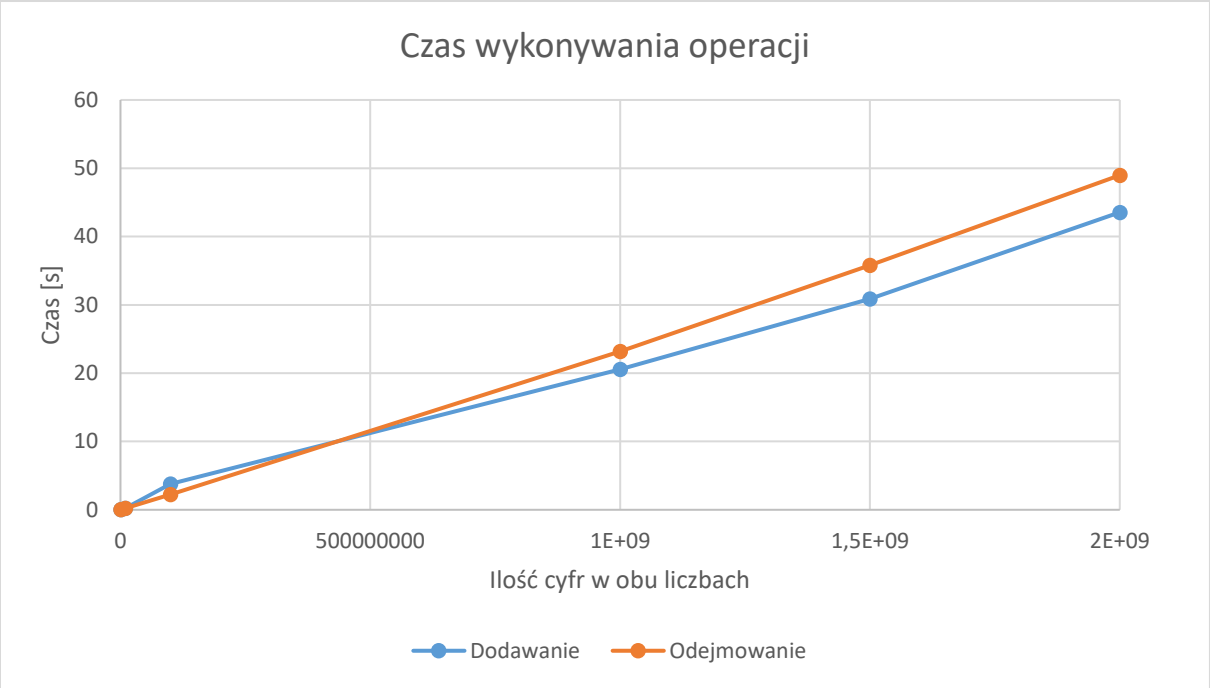
W celu zapewnienia szybszego działania programu został on skompilowany przy użyciu opcji kompilatora gcc –O1[2].

Ilość cyfr	Czas operacji [s]	Użycie pamięci RAM [GB]
1000000	0,05	
10000000	0,22	
100000000	3,79	0,6532
1000000000	20,58	6,5
1500000000	30,85	9,8
2000000000	43,55	13

Tabela 5 Dodawanie z optymalizacją.

Ilość cyfr	Czas operacji [s]	Użycie pamięci RAM
1000000	0,03	
10000000	0,22	
100000000	2,26	0,7453
1000000000	23,19	7,8
1500000000	35,78	10,8
2000000000	48,98	13,9

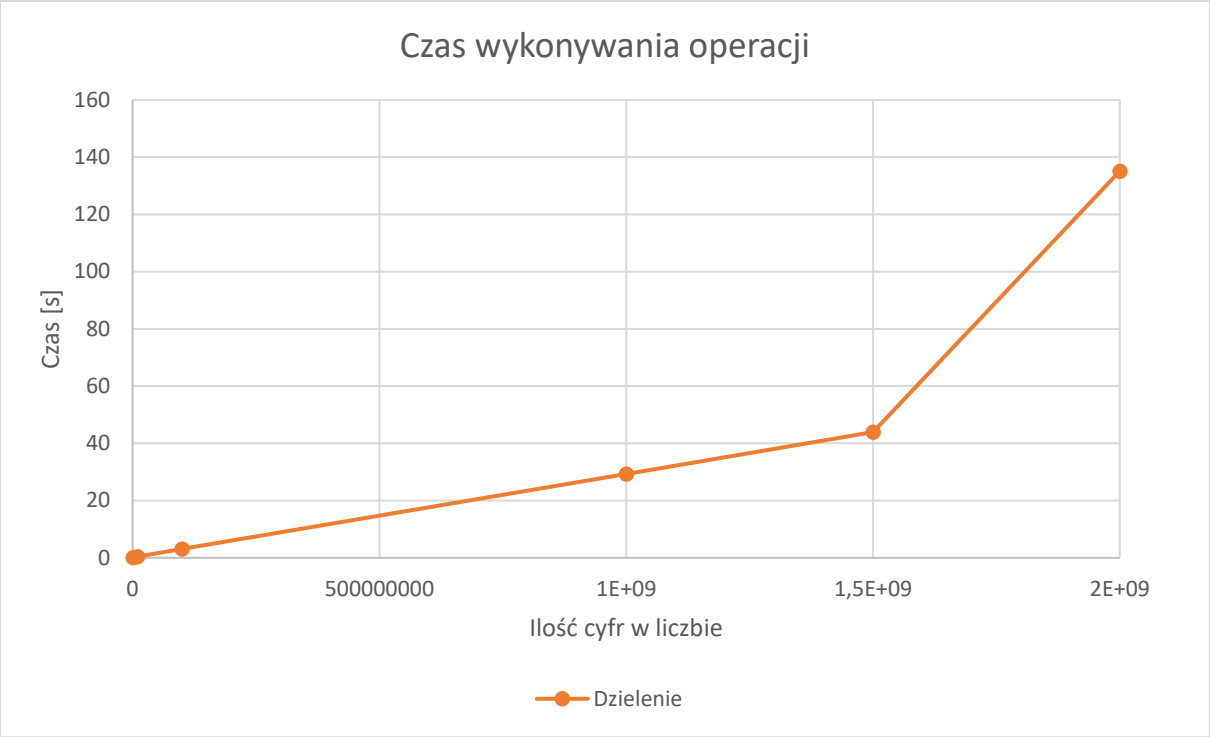
Tabela 6 Odejmowanie z optymalizacją.



Wykres 4 Czas wykonywania dodawania i odejmowania z optymalizacją.

Ilość cyfr	Czas operacji [s]	Użycie pamięci RAM
1000000	0,04	
10000000	0,41	
100000000	3,12	0,4655
1000000000	29,31	6,5
1500000000	43,92	9,8
2000000000	135,12	13

Tabela 7 Dzielenie z optymalizacją.



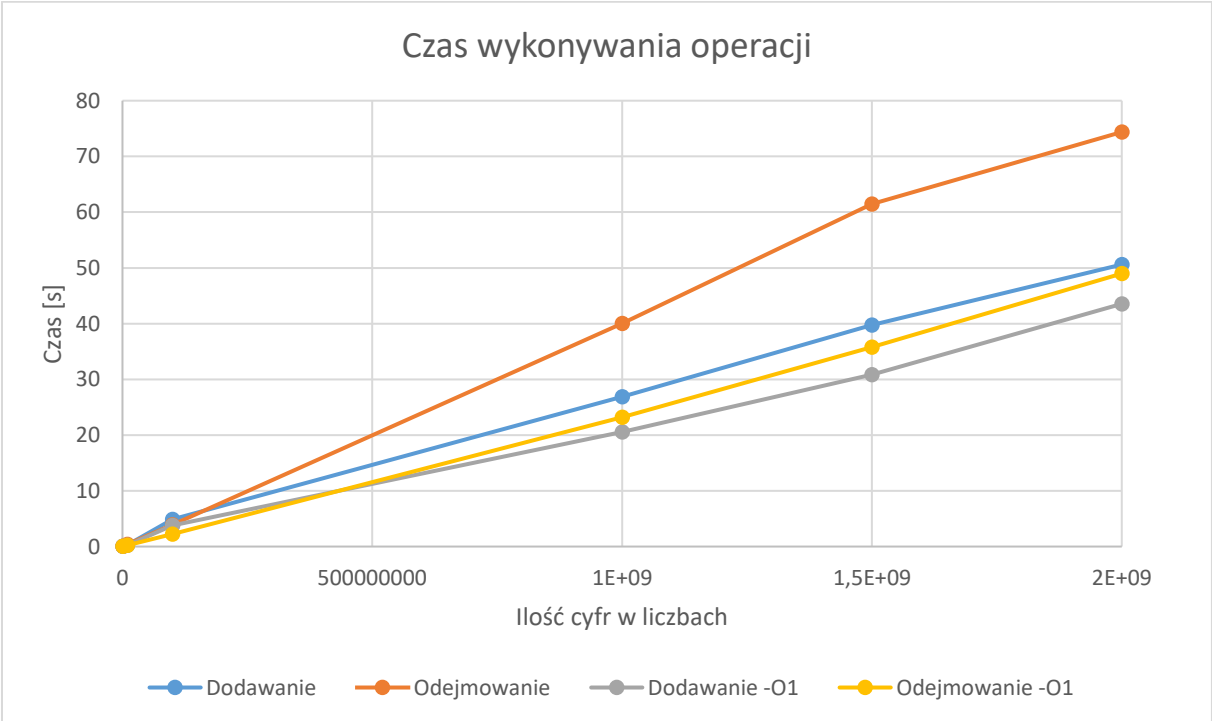
Wykres 5 Czas dzielenia z optymalizacją.

Ilość cyfr	Czas operacji [s]
3000	0,68
6250	2,83
12500	11,28
25000	45,5
50000	178,66

Tabela 8 Mnożenie z optymalizacją.

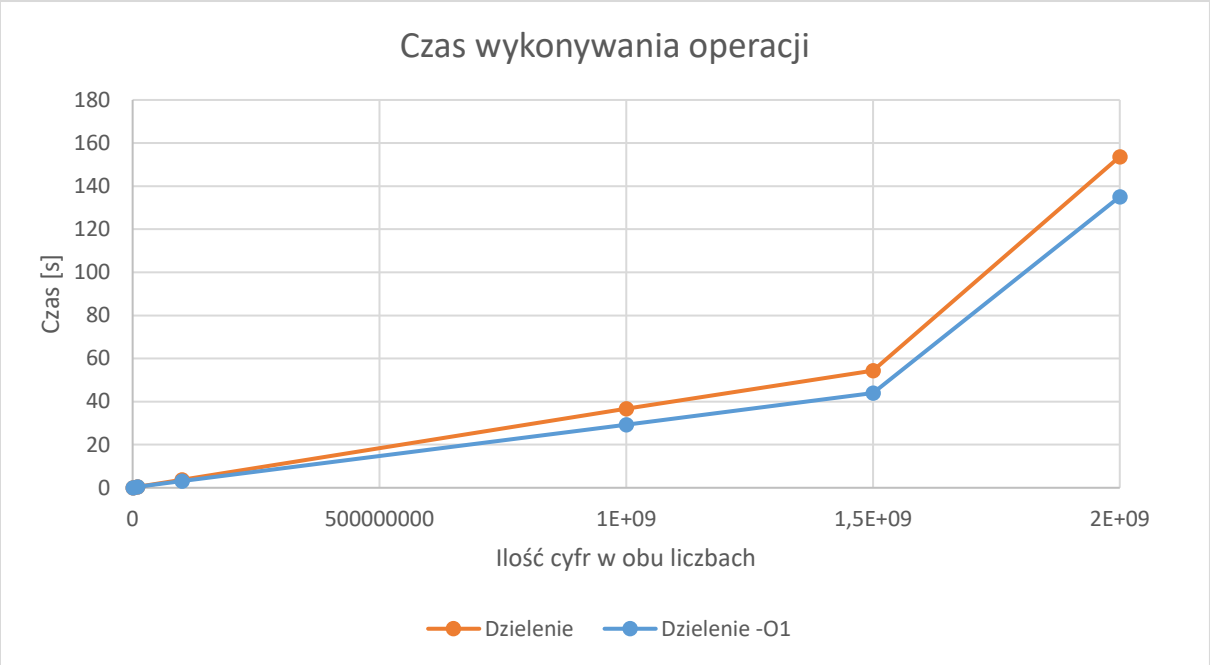
Wykres 6 Czas wykonywania operacji mnożenia z optymaliacją.

10. Porównanie wyników testów wydajnościowych.



Wykres 7 Porównanie operacji dodawania oraz odejmowania z optymalizacją i bez niej.

Na powyższym wykresie możemy zaobserwować, że optymalizacja gcc przyczyniła się do szybszego wykonywania programu. Dzięki jej zastosowaniu operacja odejmowania w górnym zakresie testów wykonywała się około 25 sekund krócej co stanowi prawie 30% całkowitego czasu wykonania.



Wykres 8 Porównanie operacji dzielenia z optymalizacją i bez niej.

Operacja dzielenia po zastosowaniu optymalizacji wykonuje się szybciej dla dużych liczb. Podczas dzielenia dwóch liczb posiadających 2.000.000.000 cyfr każda, różnica w czasie pracy programu wynosi około 20 sekund (przyspieszenie o około 12%) na korzyść kodu z optymalizacją.

Wykres 9 Porównanie operacji mnożenia z optymalizacją i bez niej.

W przypadku mnożenia, podobnie jak w przypadku dzielenia zoptymalizowany kod wykonuje się nieco szybciej. Różnica wynosi około 30 sekund (przyspieszenie o 15%) przy liczbach posiadających 50.000 cyfr każda.

Ilość cyfr	Czas operacji [s]
100	0,05
250	0,32
500	2,48
1000	17,91
1500	58,6

Tabela 9 Mnożenie metodą chłopów rosyjskich.

Ilość cyfr	Czas operacji [s]
100	0
250	0,01
500	0,04
1000	0,14
1500	0,25

Tabela 10 Mnożenie metodą iloczynów częściowych.

Wykres 10 Porównanie czasu wykonywania mnożenia metodą chłopów rosyjskich oraz iloczynów częściowych.

11.Testy jednostkowe.

Poprawność działania biblioteki była również testowana przy użyciu testów jednostkowych. W tym celu wykorzystana została biblioteka **cunit**[4] umożliwiająca realizację tego typu zadań w języku C. Po wykonaniu zestawu testów biblioteka wyświetla w konsoli tabelę podsumowującą przebieg testów. Mierzony jest również całkowity czas wykonywania testów w sekundach.

Poniżej przykładowy wynik testów:

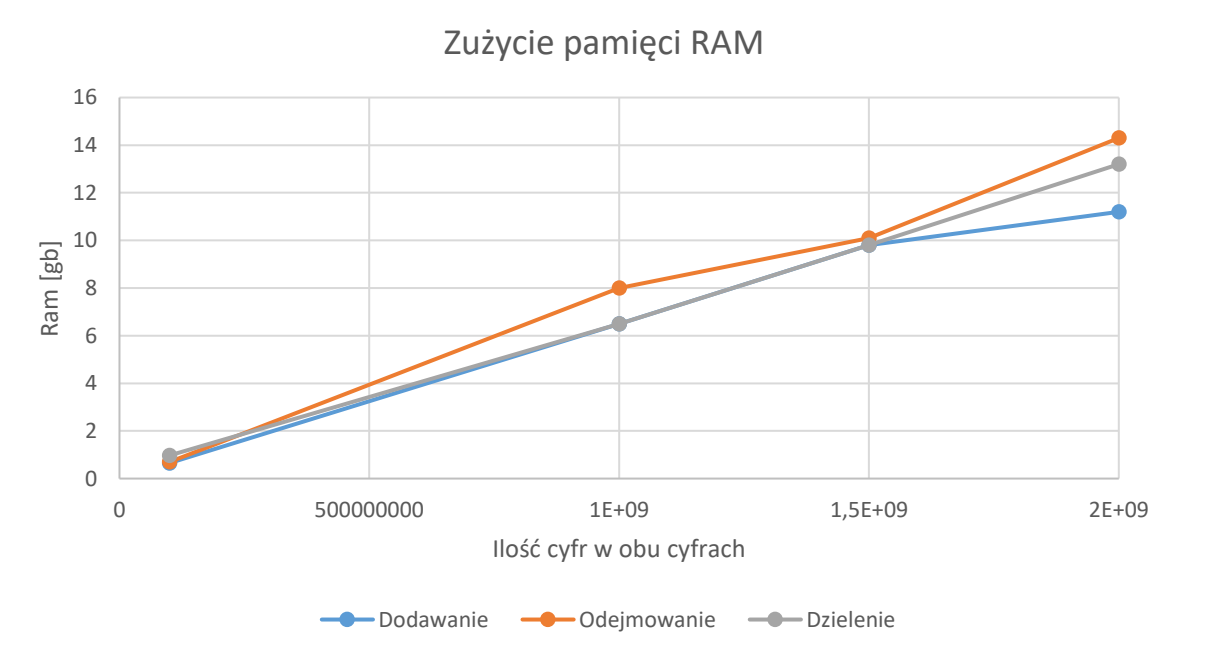
```
     CUnit - A unit testing framework for C - Version 2.1-2
     http://cunit.sourceforge.net/


Suite: unit tests
  Test: test of any_add_BIAS for given values ...passed
  Test: test of any_sub_BIAS for given values ...passed
  Test: test of any_div_BIAS for given values ...passed
  Test: test of any_mul_BIAS for given values ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      4      4      4      0        0
             asserts     21     21     21      0      n/a

Elapsed time =    0.020 seconds
```

Rysunek 6 Wynik przykładowych testów jednostkowych.

Program został również wyposażony w możliwość przeprowadzania testów manualnych poprzez uruchomienie programu i przekazanie do niego odpowiednich parametrów. Przykładowa sesja to np.: **./libtest 5 + 5**.

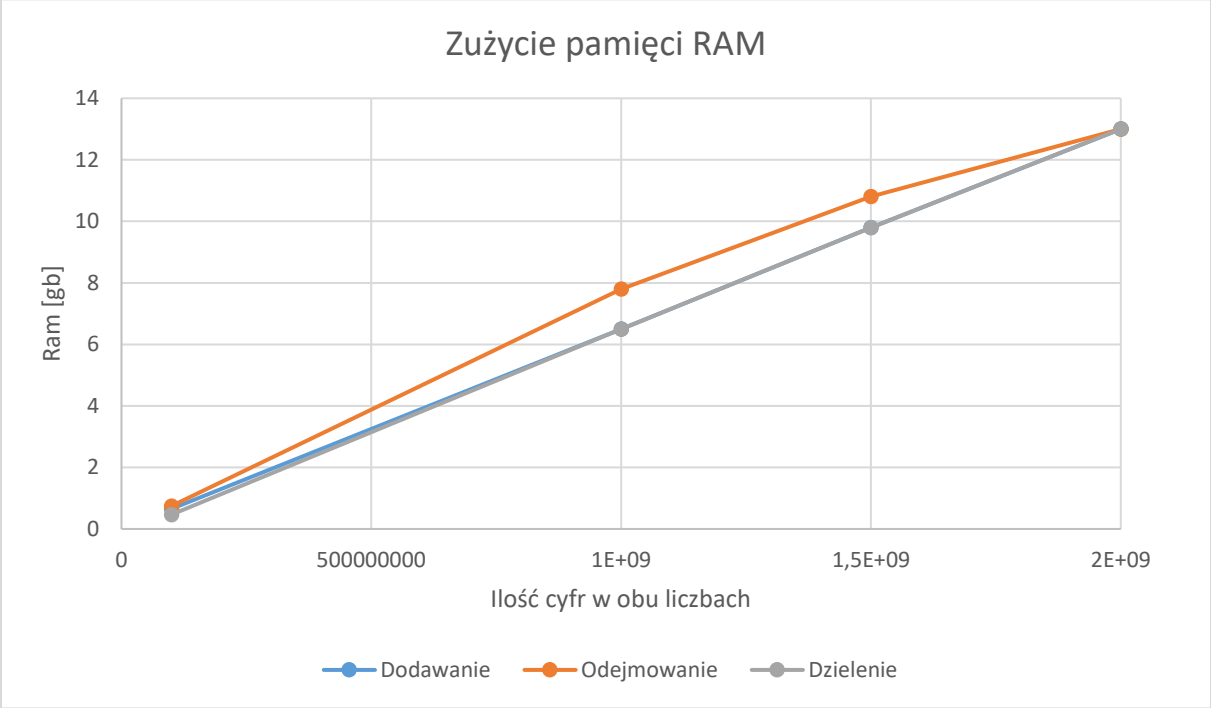
12.Zużycie pamięci RAM.

Zużycie pamięci ram zostało zbadane przy pomocy narzędzia top.



Wykres 11 Zużycie pamięci ram w operacjach dodawania, odejmowania oraz dzielenia bez optymalizacji.

Wyniki programu skompilowanego przy pomocy optymalizacji -O1 prezentują się następująco:



Wykres 12 Zużycie pamięci ram w operacjach dodawania, odejmowania oraz dzielenia z optymalizacją.

Jak możemy zauważyć na powyższych wykresach, optymalizacja kodu w tym przypadku nieznacznie obniżyła zużycie pamięci dla funkcji realizujących odejmowanie oraz dzielenie.

13.Wnioski.

Realizacja biblioteki operującej na liczbach dowolnej precyzji pozwoliła na zapoznanie się z wieloma przydatnymi w programowaniu narzędziami. Zaimplementowane metody pozwalają na wykonywanie obliczeń w zadowalającym czasie.
Przeprowadzone testy jednostkowe, manualne oraz testy na wycieki pamięci nie wykazały żadnych błędów.
Liniowa zależność czasu wykonywania i zużycia pamięci w funkcji długości liczby dla operacji dodawania oraz odejmowania.
Dłuższe czasy dla operacji odejmowania w stosunku do dodawania wynikają z bardziej złożonej obsługi przeniesienia.
Pomimo zastosowania sum częściowych zarówno w mnożeniu jak i dzieleniu operacja realizująca mnożenie okazała się o wiele wolniejsza.
Zastosowanie optymalizacji kompilatora gcc przyniosło zyski dla wszystkich testowanych metod. Największa poprawa nastąpiła dla funkcji realizującej odejmowanie.
Profilowanie kodu przy użyciu narzędzia perf wykazało, że najwięcej czasu program spędza w funkcjach assemblerowych wykonujących operację dodawania oraz odejmowania. Dzieje się tak, ponieważ właśnie w tych funkcjach realizowane są najważniejsze obliczenia związane z arytmetyką biblioteki.

Dzięki monitorowaniu kodu narzędziem valgrind udało się wychwycić i wyeliminować wycieki pamięci.
Operacja mnożenia została początkowo zaimplementowana, jako wielokrotne dodawanie. W tej postaci była jednak bardzo powolna. W celu zwiększenia szybkości operacji mnożenia zaimplementowana została również metoda szybkiego mnożenia metodą chłopów rosyjskich[7]. Niestety jej implementacja nie przyniosła oczekiwanych rezultatów i pomysł ten został porzucony. Ostateczna metoda, która została przyjęta opiera się na sumowaniu iloczynów częściowych.

14.Literatura:

- [1] IA-32 Intel® Architecture Software Developer's Manual Volume 2: Instruction Set Reference
- [2] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- [3]http://jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl/ProgramowanieWspolbiezne/lab/LabLinux56.pdf?fbclid=IwAR2AQQw8_4gh76t30VK5afH47foJBKkeTN_MQUW_FvfF1cSemV8DtBxR8VA [str. 26-27]
- [4] http://cunit.sourceforge.net/
- [5]https://dev.to/etcwilde/perf---perfect-profiling-of-cc-on-linux-of
- [6]http://www.valgrind.org/docs/manual/manual.html
- [7] http://thedailywtf.com/articles/Programming-Praxis-Russian-Peasant-Multiplication