Implementacja biblioteki arytmetyki liczb stałoprzecinkowych dowolnej precyzji z wykorzystaniem wewnętrznej reprezentacji U2

Kamil Pawelski

Adam Troszczyński

ITE Wałbrzych

2022/2023

Spis treści

I. CEL PROJEKTU

Celem tego projektu była implementacja biblioteki umożliwiającej operacje arytmetyczne na liczbach stałoprzecinkowych dowolnej precyzji, opartej na wewnętrznej reprezentacji w systemie U2. Biblioteka miała za zadanie realizować podstawowe operacje matematyczne, takie jak dodawanie, odejmowanie, mnożenie i dzielenie. Wszystkie te operację miały być zoptymalizowane, wykorzystując możliwości procesora. Następnie przeprowadzono serię testów w celu oceny wydajności naszego rozwiązania.

II. ZAŁOŻENIA I OGRANICZENIA

Założenia projektowe:

- Obsługa dowolnej precyzji: Biblioteka powinna umożliwiać manipulację liczbami stałoprzecinkowymi o dowolnej precyzji.
- Wewnętrzna reprezentacja U2: Implementacja powinna wykorzystywać wewnętrzną reprezentację U2 do przechowywania liczb stałoprzecinkowych.
- Wspomaganie assemblera: Biblioteka powinna wykorzystywać kod assemblera w celu zoptymalizowania operacji arytmetycznych.
- Kompatybilność z GNU GCC: Implementacja biblioteki powinna być kompatybilna z GNU GCC, w systemach opartych na Linuxie.

Ograniczenia:

- Zależność od architektury procesora: Wykorzystanie assemblera może wprowadzać zależność od konkretnej architektury procesora. Biblioteka może działać nieoptymalnie lub nie być kompatybilna z niektórymi architekturami procesorów.
- Dostępność instrukcji assemblera: Biblioteka może być ograniczona przez dostępność odpowiednich instrukcji assemblera w danym środowisku lub architekturze procesora. Niektóre instrukcje mogą być dostępne tylko w nowszych wersjach procesorów.

III. TECHNOLOGIE

Zastosowane języki:

Nasz projekt został zrealizowany przy użyciu języka C++ oraz języka assemblera. Wybór języka C++ był dla nas naturalny, ponieważ mieliśmy już doświadczenie w tym języku z poprzednich kursów na studiach. W porównaniu do potencjalnego wyboru języka C, decyzja ta przyniosła nam wiele korzyści. Jedną z największych zalet użycia języka C++ było skorzystanie z gotowych szablonów z biblioteki STL. Dzięki temu uniknęliśmy konieczności samodzielnego zarządzania dynamicznie alokowaną pamięcią, co jest często problematyczne w jeżyku C. Szablony z biblioteki STL dostarczyły nam gotowe mechanizmy i struktury danych, które znacząco ułatwiły pracę.
Nasza biblioteka została napisana z myślą o architekturze 32-bitowej.

Oprogramowanie:

Nasza biblioteka została z powodzeniem napisana i przetestowana na dystrybucji Linuxa Ubuntu. W celu zapewnienie jakości i poprawności działania, samodzielnie stworzyliśmy testy jednostkowe, które pozwoliły nam przetestować poszczególne działania.
Do kompilacji biblioteki wykorzystaliśmy kompilator GCC (GNU Compiler Collection). Wykorzystanie GCC zapewniło nam kompatybilność z różnymi systemami operacyjnymi opartymi na Linuxie.

IV. REALIZACJA PROJEKTU

Na samym początku napotkaliśmy pewne trudności. Nasze początkowe próby implementacji biblioteki były utrudnione z powodu niepełnego zrozumienia jej dokładnego działania. W rezultacie, nasz postęp w pisaniu programu był ograniczony. Postanowiliśmy rozpocząć od implementacji struktury naszych liczb, której reprezentacja zostanie opisana w kolejnym punkcie. Skupiliśmy się przede wszystkim na konstruktorze, aby upewnić się, że reprezentacja liczby będzie dla nas wystarczająco prosta do późniejszego używania w operacjach. Kolejnym krokiem było dodanie metody umożliwiającej wypisywanie naszej liczby w formacie U2, co umożliwiło nam późniejsze testowanie. Po zaimplementowaniu tej funkcjonalności przystąpiliśmy do realizacji kolejnych działań arytmetycznych, takich jak dodawanie, odejmowanie, mnożenie i dzielenie. Te etapy przyniosły nam wiele trudności, które zostaną opisane w dalszej części. Po zakończeniu implementacji wszystkich operacji, mogliśmy skupić się na przeprowadzeniu badań wydajnościowych, aby ocenić działanie naszej biblioteki.

V. OPIS KODU

Reprezentacja liczby:

Utworzyliśmy klasę o nazwie TC, która obejmuje dwa pola, tworzące kompletność naszej liczby.

- Pole vector<uint8_t> _number stanowi kontener, w którym przechowujemy liczbę reprezentowaną w systemie U2. Każdy indeks w tym kontenerze zawiera 8-bitową część liczby.
- Pole int _position określa pozycję najmniej znaczącego bitu liczby.

Konstruktor :

Konstruktor nie tylko zapisuje liczbę i pozycję podaną przez użytkownika, ale również przekształca liczbę w kontenerze. Przyjęliśmy określone przedziały, które mogą być umieszczone pod indeksem w kontenerze. W przypadku, gdy podana przez użytkownika liczba różni się od ostatniej liczby w danym przedziale, wykonujemy dodatkowe operacje przesunięcia.

(Pierwsza pozycja w danym przedziale, Ostatnia pozycja w danym przedziale)

…	(15,8)	(7,0)	(-1,-8)	…

Na przykład, jeśli użytkownik poda 8-bitową liczbę i pozycję 0 konstruktor nie wykonuje dodatkowych operacji i zapisuje ją dokładnie tak, jak została podana. Ta liczba będzie przechowywana w jednym indeksie kontenera.

	Pozycja	Liczba w kontenerze
Użytkownik	0	10001001
Konstruktor	0	10001001

Natomiast, jeśli użytkownik poda tę samą 8-bitową liczbę, ale pozycję 3, przedział dla tej liczby będzie wyglądał (10,3), co nie pasuje do naszego zamysłu. W takim przypadku konstruktor rozpocznie operacje przesunięcia, aby nasza liczba rozpoczynała się od określonych przedziałów. W tym przypadku liczba zostanie zapisana na dwóch indeksach w naszym kontenerze (15,8) i (7,0).

	Pozycja	Liczba w kontenerze
Użytkownik	3	10001001 (10001001000)
Konstruktor	0	11111000 1001000

Wypisywanie liczby:

Liczba jest zapisywana bit po bicie za pomocą operacji logicznych do stringa. W zależności od określonych warunków, wykonują się dodatkowe operację, głównie polegające na dodawaniu zera lub przecinka.

Ogólne zasady operacji:
Podczas wykonywania operacji arytmetycznych napotkaliśmy wiele problemów, które zostaną opisane w późniejszych punktach. Początek każdej z tych operacji jest praktycznie identyczny. Rozpoczynamy od usunięcia przecinka, jeśli taki występuje, poprzez przesunięcie obydwu liczb.

Dodawanie i Odejmowanie:
W przypadku operacji dodawania i odejmowanie rozpoczynamy od utworzenia nowego kontenera, w którym zostanie zapisany wynik działania. Pozycja wyniku będzie przyjmować mniejszą pozycję spośród dwóch podanych liczb. Dodatkowo, zmienna oznaczająca pozycję najstarszego bitu przyjmie większą pozycję spośród obu liczb. Korzystając z tych dwóch zmiennych, obliczamy rozmiar kontenera wynikowego.
Następnie obliczane są indexy, aby wiedzieć, od którego miejsca w kontenerze należy dodawać lub odejmować. Operację te są wykonywane na liczbach dodatnich. W przypadku, gdy mamy do czynienia z liczbą ujemną, zamieniamy ją na jej odpowiednik dodatni. Następnie przekazujemy te liczby do odpowiednich funkcji dodawania lub odejmowania, które są napisane w asemblerze. Wynik jest następnie zapisywany w systemie U2.

Mnożenie:
Podczas operacji mnożenia, rozmiar wynikowego kontenera jest wyliczany na podstawie sumy rozmiarów obu liczb. Następnie, za pomocą dwóch pętla, obliczamy kolejne iloczyny częściowe, które są dodawane do kontenera wynikowego.

Dzielenie:
Dzielenie zostało wykonane za pomocą algorytmu dzielenia nieodtwarzającego. Aby zapewnić, że dzielna jest co najmniej o 8 bitów większa od dzielnika, przeprowadzamy odpowiednie rozszerzenie.
Rozpoczynamy algorytm dzielenia nieodtwarzającego poprzez przeniesienie części dzielnej o rozmiarze dzielnika do nowego kontenera. Następnie, po wykonaniu odpowiednich operacji dodawania lub odejmowania, przesuwamy nasz nowy kontener o 1 bit w lewo, powtarzając ten proces aż do ostatniego bitu dzielnej. Na sam koniec sprawdzamy czy wynik, który posiadamy aktualnie jest prawidłowy mnożąc go przez dzielnik. Jeśli tak to kończymy działanie, jeśli nie to wykonujemy poprzedni algorytm odejmowania/dodawania 8 razy.

VI. TESTY POPRAWNOŚCI

Wszystkie działania arytmetyczne w bibliotece zostały objęte kompleksową serią testów, które obejmują różnorodne przypadki użycia tych metod. Testy te służą do zapewnienia poprawności działania funkcji oraz umożliwiają szybkie sprawdzeniem czy wprowadzone zmiany nie zakłóciły poprawności wcześniej zaimplementowanych algorytmów.
Z uwagi na opisane poniżej problemy, postanowiliśmy stworzyć własną bibliotekę służącą dedykowaną bibliotekę do testowania naszych metod. Zawiera ona:

- isPassed(TC,TC,TC,char,std::string): Ta metoda przyjmuje jako argumenty pierwszą liczbę, drugą liczbę, oczekiwany wynik operacji, znak operacji oraz nazwę testu. Jej celem jest sprawdzenie, czy dany test zakończył się powodzeniem.
- setAutoTest(): Ta metoda uruchamia serię testów, które zostały zaimplementowane wcześniej w kodzie.
- manualTest(TC,TC,char,TC): Metoda ta umożliwia ręczne wprowadzanie oraz przeprowadzanie testów. Przyjmuje jako argumenty pierwszą liczbę, drugą liczbę, znak operacji oraz oczekiwany wynik operacji.

Zauważone błędy w czasie testowania
- Wykonywanie operacji dzielenia działa w wypadku gdy obie pozycje są ustawione na 0. W innym wypadku nasze dzielenie w obecnej implementacji nie potrafi sobie poradzić z innymi danymi.
- Ujemne liczby w działaniach zwracały błędne wyniki.
- Algorytm źle intepretował liczbę po przecinku.

VII. Badania wydajnościowe

Specyfikacja techniczna komputera:

Badania zostały wykonane na systemie operacyjnym Ubuntu 22.04.2, GCC 12.2.0.

Założenia początkowe:

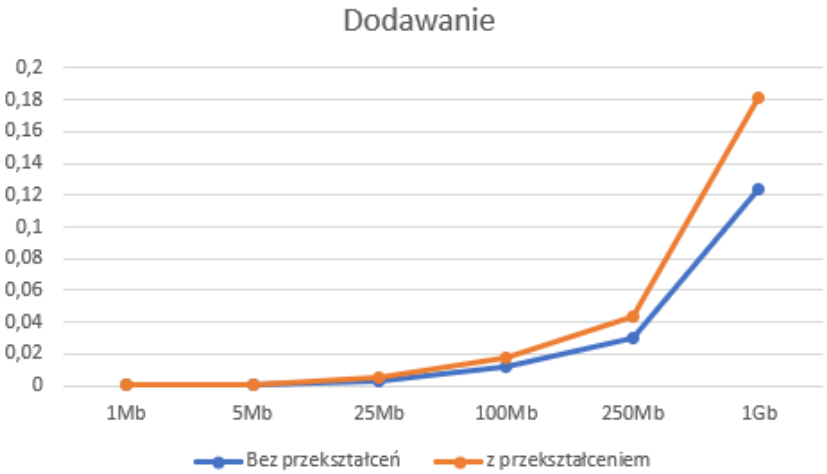
- Przeprowadzono badania na losowo wygenerowanych danych, wykorzystując srand do inicjalizacji generatora liczb pseudolosowych.
- Czas badania został zmierzony przy użyciu biblioteki time.h.
- Badania zostały przeprowadzone dla operacji arytmetycznych bez przekształcania liczby oraz z uwzględnieniem przekształcania.
- Operację dodawania, odejmowania i mnożenia zostały wykonane na danych o rozmiarach: 1Mb, 5Mb, 25Mb, 100Mb, 250Mb oraz 1Gb. W przypadku mnożenia, mnożnik zawsze ma rozmiar równy 1Kb.
- Operacja dzielenia została przeprowadzona na danych o rozmiarach: 2Kb, 10Kb, 25Kb, 50Kb, 100Kb, oraz 500Kb, przy czym dzielnik zawsze ma rozmiar równy 10Kb.
- Wyniki pomiarów są przedstawione jako średnia ze 100 wykonanych pomiarów, z wyjątkiem mnożenia (100Mb,250Mb,1Gb) oraz dzielenia(100Kb,500Kb) ze względu na długi czas obliczeń.
- Wyniki są podane w jednostce czasu – sekundach.

Wyniki:

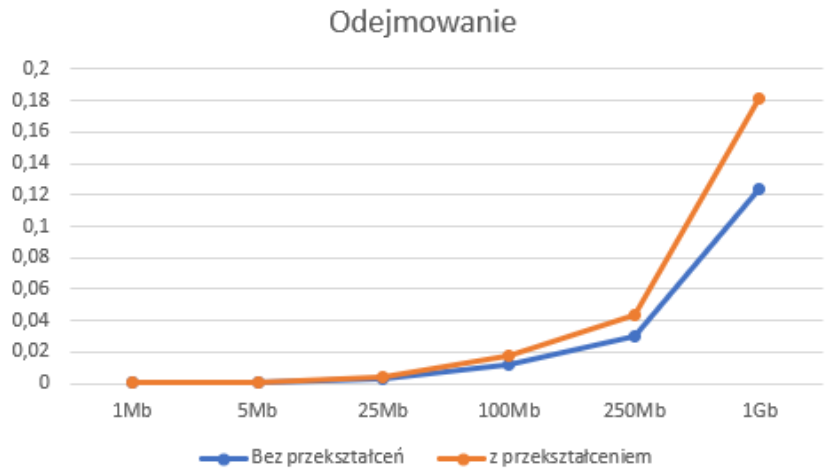
	Dodawanie Przekształcenia/bez			Odejmowanie Przekształcenia/bez			Mnożenie Przekształcenia/bez	
1Mb	0.00013122	0.00018113		0.00013244	0.0001809		0.05209152	0.05250157
5Mb	0.0006536	0.00094077		0.00063363	0.00092419		0.26458905	0.26488898
25Mb	0.00328475	0.00492856		0.00301923	0.00443319		1.2619775	1.2728982
100Mb	0.011868	0.017519		0.011943	0.017642		5.298987	5.342893
250Mb	0.029659	0.043799		0.029693	0.043823		13.30988	13.42162
1Gb	0.123409	0.181358		0.123392	0.181462		55.0621	55.1228

	Dzielenie Przekształcenia/bez				Dodawanie Z assemblerem/bez	
1Kb	0.004959	0.005265		10Mb	0.0012983	0.0099684
10Kb	0.001816	0.001852		50Mb	0.0074125	0.0499567
25Kb	0.456662	0.456724		100Mb	0.0140617	0.0982847
50Kb	1.21244	1.21251		250Mb	0.0336577	0.2509198
100Kb	2.48155	2.48169		500Mb	0.0719415	0.4953222
500Kb	14.8116	14.8117		1Gb	0.1331696	1.0023667

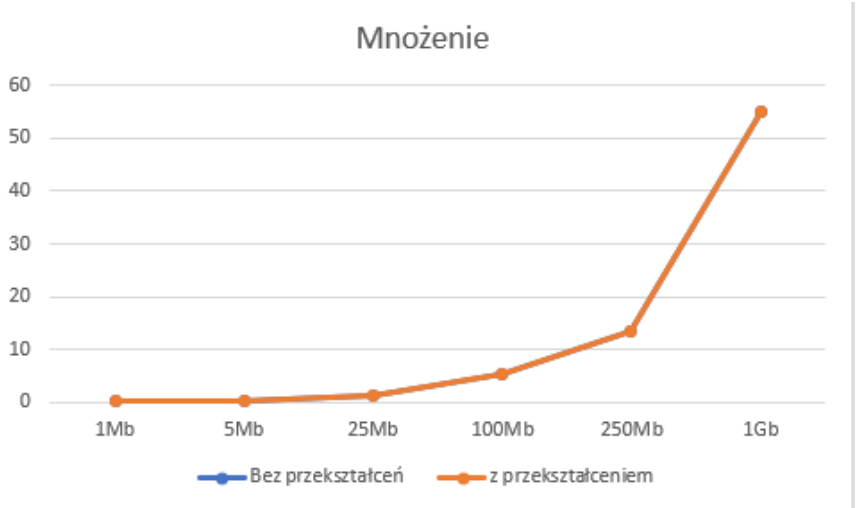
Analiza wyników



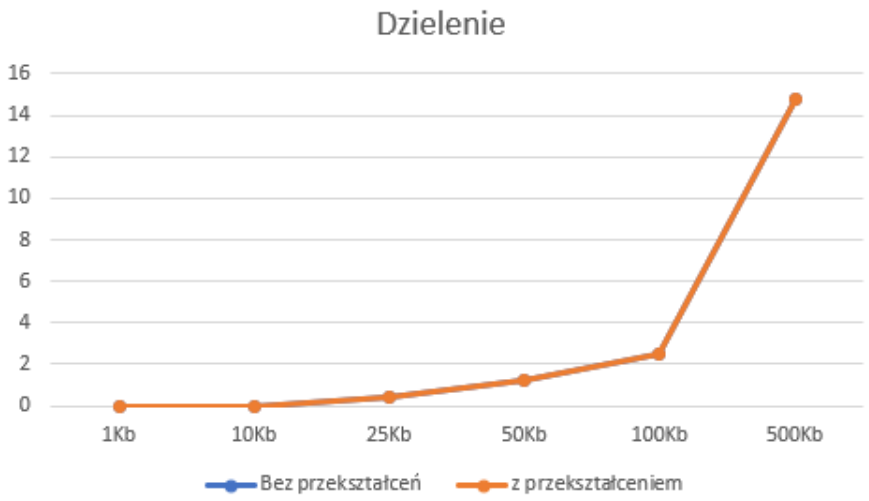
Z analizy przedstawionego wykresu, który porównuje czas dodawania uwzględniając przekształcenie liczby oraz czas samego dodawania, można dostrzec, że dla dużych rozmiarów danych zaczyna się pojawiać widoczna różnica czasowa.



Powyższy wykres przypomina poprzedni, jednak tym razem można zauważyć nieco większą różnicę w czasach.

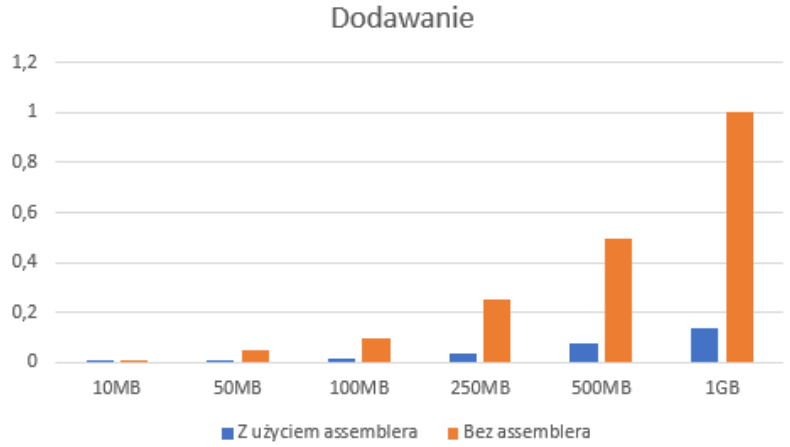


Na powyższym wykresie zaobserwować można prawie niezauważalną różnicę czasową, która wynika z zastosowania skomplikowanego algorytmu, wymagającego dużo czasu do przetwarzania



Na wykresie dzielenia możemy dostrzec analogiczną sytuację jak w przypadku mnożenia. Ze względu na długotrwałe przetwarzanie obliczeń na dużych zbiorach danych (1 Gb), nie jesteśmy w stanie zauważyć, czy występuje zjawisko podobne jak przy dodawaniu lub odejmowaniu.

W operacjach mnożenia i dzielenia obserwujemy znaczący wzrost czasu wykonania, gdy pracujemy z większymi zbiorami danych, w porównaniu do operacji dodawania lub odejmowania. To zjawisko wynika z konieczności przeprowadzenia znacznie większej liczby operacji w tych przypadkach.



Wykres przedstawia porównanie średniego czasu wykonywania dodawania z użyciem assemblera i bez jego użycia. Wyniki jednoznacznie wskazują, że zastosowanie assemblera prowadzi do zauważalnie szybszego czasu wykonania. Im większy rozmiar liczby, tym różnica w czasie jest jeszcze bardziej wyraźna. Oznacza to, że operację, które korzystają z assemblera są bardziej zoptymalizowane i wydajniejsze.

VIII. NAPOTKANE PROBLEMY

Pierwszym wyzwaniem, które napotkaliśmy, dotyczyło pozycji liczby przy przekazywaniu jej dalej. Aby ułatwić operacje na konkretnej liczbie, musieliśmy wprowadzić specjalne przesunięcia. Problem ten został rozwiązany poprzez zmianę koncepcji zapisu liczby, co opisywaliśmy w konstruktorze.

Kolejnym problemem był błędny wynik operacji, szczególnie w przypadku ujemnych liczb. Aby temu zaradzić, postanowiliśmy przekazywać ich odpowiedniki dodatnie i dokonywać odpowiednich operacji dodawania lub odejmowania, w zależności od znaków liczby. Ta zmiana znacząco poprawiła rezultaty.

Ostatnim problemem, z jakim się zmierzyliśmy, było przeprowadzanie operacji na liczbach z częścią dziesiętną. Ponownie, wyniki nie zgadzały się z oczekiwanymi. Aby rozwiązać ten problem, zdecydowaliśmy się na operację przesunięcia, aby pozbyć się części dziesiętnej. Dzięki temu nasze liczby wychodziły zgodnie z oczekiwaniami.

Mieliśmy również trudności z gotowymi bibliotekami do testowania. Wybrane przez nas biblioteki wymagały kompilacji w wersji 64-bitowej, podczas gdy nasz kod był przeznaczony dla architektury 32-bitowej. Dlatego postanowiliśmy napisać własną bibliotekę testową dla naszego kodu.

Naprawa problemów związanych z dzieleniem stanowiła duże wyzwanie. Początkowo mieliśmy trudności nawet z podejściem do tego zagadnienia. Dzięki pomocy na zajęciach dowiedzieliśmy się, że możemy rozszerzyć nasze liczby i udało nam się zaimplementować algorytm dzielenia nieodtwarzający.

Podany wcześniej problem z dzieleniem nie został niestety naprawiony, ponieważ już zabrakło czasu na rozwiązanie owego problemu. Jedynie co można zrobić to jak pisaliśmy podawać dla obu liczb pozycję 0.

IX. PODSUMOWANIE I WNIOSKI

Celem naszego projektu było stworzenie biblioteki, która umożliwiałaby wykonywanie operacji arytmetycznych na liczbach stałoprzecinkowych w systemie U2. Biblioteka pozwala na podstawowe operacje arytmetyczne. Niestety nie udało się osiągnąć pełnej skuteczności w obliczeniach jak w przypadku dzielenia. Wnioskiem z naszego projektu jest, że zastosowanie funkcji assemblera w bibliotece arytmetyki liczb stałoprzecinkowej dowolnej precyzji w systemie U2 przynosi znaczące korzyści pod względem wydajności. Dzięki temu możemy osiągnąć szybsze i bardziej efektywne operacje arytmetyczne, szczególnie w przypadku dużych rozmiarów liczb. Jest to istotne dla aplikacji, które wymagają skutecznego przetwarzania danych i minimalizacji czasu obliczeń

X. NASZE ZAŁOŻENIA I POTENCJALNE ZMIANY

- **Zakres i typ:**
  W naszym projekcie przyjęliśmy, że biblioteka obsługuje liczby stałoprzecinkowe dowolnej precyzji w systemie U2. Istnieje możliwość zaimplementowania funkcji konwersji między innymi systemami liczbowymi, aby zwiększyć elastyczność biblioteki.
- **Wydajność:**
  Choć zauważyliśmy znaczącą poprawę wydajności przy użyciu assemblera, nadal istnieje potencjał do dalszych optymalizacji. Możemy zbadać alternatywne metody optymalizacji kodu lub skorzystać z bardziej zaawansowanych technik.
- **Obsługa błędów:**
  W naszej bibliotece wprowadziliśmy podstawową obsługę błędów, takich jak dzielenie przez zero. Jednak warto rozważyć rozbudowę systemu obsługi błędów, aby zapewnić bardziej szczegółowe informacje o rodzaju i przyczynie błędu.
- **Testowanie**
  Ważnym aspektem projektu jest rozbudowa zestawu testów jednostkowych. Możemy rozszerzyć zestaw testów, aby uwzględnić różnorodne scenariusze i przypadki testowe

XI. KOMPILACJI

Kod dostępny na githubie https://github.com/Amanowsky/U2Math_v2.git

W celu kompilacji musimy zainstalować bibliotekę:
Sudo apt-get install gcc-multilib g++-multilib

Kompilacja odbywa się poleceniem „ gcc -m32 -g main.cpp TC.cpp vectorAdd.s vectorSub.s vectorMul.s -m main -lstdc++ „ pozwala na skompilowanie programu, który składa się z plików źródłowych main.cpp TC.cpp oraz asemblerowych plików vectorAdd.s, vectorSub.s i vectorMul.s

- **-m32** wskazuje na to, że kompilacja ma być przeprowadzona dla architektury 32-bitowej.
- **-g** umożliwia dołączenie informacji debugowania do wygenerowanego pliku wynikowego.
- **-o main** określa nazwę pliku wynikowego jako „main”
- **-lstdc**++ włącza linkowanie biblioteki standardowej języka C++

W rezultacie kompilacji zostanie utworzony plik wykonywalny o nazwie „main”.

XII. PODZIAŁ PRACY

Struktura liczby – Kamil Pawelski i Adam Troszczyński
Konstruktor – Kamil Pawelski
Algorytm dodawania – Kamil Pawelski
Algorytm odejmowania – Kamil Pawelski
Algorytm mnożenia – Adam Troszczyński
Algorytm dzielenia – Kamil Pawelski
Biblioteka do testów – Adam Troszczyński
Napisane testy – Adam Troszczyński
Metody pomocnicze – Kamil Pawelski i Adam Troszczyński
Badania – Adam Troszczyński
Raport – Kamil Pawelski i Adam Troszczyński