Biblioteka arytmetyki liczb stałoprzecinkowych dowolnej precyzji z wykorzystaniem wewnętrznej reprezentacji U2.

Leszek Błażewski

Karol Noga

Semestr letni 2018/2019

Spis treści

1 Założenia projektowe

Celem projektu była implementacja biblioteki oferującej podstawowe operacje arytmetyczne na liczbach dowolnej precyzji, która wykorzystuje system uzupełnień do dwóch jako bazowy. Jednym z wymagań projektowych było zapewnienie możliwe najszybszej oraz najbardziej optymalnej implementacji danych algorytmów wykorzystując do tego celu instrukcje procesora. Dalsza część projektu obejmowała sprawdzenie poprawności zaimplementowanych operacji oraz przeprowadzenie testów wydajnościowych. Ostatnim etapem była analiza napisanego kodu przy pomocy narzędzi oferujących funkcje pozwalające na zmierzenie wydajności danych kawałków kodu, profilowanie oraz analizę dotyczącą zarządzania pamięcią.

2 Technologie i narzędzia

Rozdział ten traktuje o argumentach, które skłoniły nas do wyboru danego języka oraz zawiera krótki opis narzędzi i technologi, które wykorzystaliśmy w projekcie.

Cały projekt wraz z dokumentacją dostępny jest w publicznym repozytorium, które dostępne jest pod poniższym linkiem:

https://github.com/dex1g/OiAK

2.1 Wybrany język

Zdecydowaliśmy się na implementację biblioteki w języku C, ponieważ kod programu kompilowany jest do kodu natywnego, dzięki czemu algorytmy pisane w tym języku przy odpowiedniej implementacji są wydajniejsze niż te pisane w językach interpretowanych oraz uruchamianych na maszynach wirtualnych. Kolejnym powodem wyboru tego języka jest łatwość integracji instrukcji mikroprocesora z kodem pisanym w tym języku oraz możliwość bezpośredniej deasemblacji kawałków kodu pisanych w C. Kluczowa jest również pełna kontrola programisty nad alokowaną pamięcią programu, co przekłada się na wydajność pisanych algorytmów. Zdecydowaliśmy się pisać całą aplikację na architekturę 32 bitową, ponieważ mamy z nią styczność na laboratoriach i wykładzie oraz znaliśmy już podstawowe założenia ABI, dzięki czemu cała implementacja przebiegła sprawniej.

2.2 Wykorzystane narzędzia

Bibliotekę staraliśmy się tworzyć w konwencji Test Driven Deployment, dzięki czemu każdy algorytm pokryty jest serią testów, która sprawdza większość możliwych przypadków oraz poprawność implementacji. Do realizacji tego zadania posłużyliśmy się biblioteką $Unity$, która oferuje bogaty zestaw funkcji pozwalających na asercję wybranych danych po przeprowadzeniu odpowiednich działań.. Podczas profilowania posłużyliśmy się narzędziem $Gprof$, które pozwoliło nam na dynamiczną analizę kodu programu oraz wykrycie funkcji, które wymagają refaktoryzacji oraz optymalizacji. Ostatnim użytym narzędziem był $Valgrind$, który dostarczył informacji o błędnym zarządzaniu pamięcią. Do projektu dodano również automatyczne budowanie testów jednostkowych przy każdej zmianie w repozytorium, zrealizowane przy pomocy platformy $Travis\ CI$. Poniżej zamieszczono listę wykorzystanych narzędzi wraz z ich zastosowaniem.

- *gcc* - Kompilacja oraz optymalizacja
- *Unity* - biblioteka użyta to implementacji testów jednostkowych
- *Gprof* - profilowanie
- *Valgrind* - analiza wykorzystanej pamięci
- *Git* - system kontroli wersji
- *Travis CI* - automatyczne budowanie projektu

3 Opis funkcjonalności biblioteki

Biblioteka oferuje cztery podstawowe operacje arytmetyczne: dodawanie, odejmowanie, mnożenie oraz dzielenie. Wszystkie operacje wymagają argumentów w formacie, który został stworzony specjalnie na potrzeby odpowiedniej reprezentacji liczb w języku *C*, w związku z czym biblioteka oferuje również zestaw funkcji pozwalający na konwersję zadanej liczby z systemu szesnastkowego na wymagany. Dodatkowo dodano możliwość odczytu danych z pliku tekstowego zawierającego liczbę szesnastkową lub bezpośrednio z pliku binarnego.

4 Wykorzystanie biblioteki

Rozdział ten zawiera informacje, które pozwalają zaimportować bibliotekę do projektu oraz sposób przeprowadzenia testów.

4.1 Dołączenie biblioteki do projektu

Aby skorzystać z wszystkich funkcji oferowanych przez bibliotekę do projektu dołączyć należy wszystkie pliki znajdujące się w folderze *src*. Następnie w plikach programu zaimportować należy pliki nagłówkowe oraz podczas kompilacji załączyć odpowiadające im pliku źródłowe oraz plik z kodem asemblerowym, które zawierają algorytmy danych operacji zrealizowane przy pomocy instrukcji procesora.

Podczas wykorzystania funkcji pamiętać należy, że wszystkie argumenty każdej z operacji są dealokowane w trakcie procesu obliczania wyniku danego działania, dlatego wskaźnik na tablicę zawierającą liczbę znajdujący się w obiekcie struktury *TCNumber* musi wskazywać na tablicę, która alokowana była dynamicznie. Gdy chcemy wykorzystać funkcje dla tablic statycznych musimy wcześniej wykorzystać funkcję *createTCNumber*, która zaalokuje wymagany obszar i przepisze do niego zawartość liczby, lecz należy liczyć się z ograniczeniami dotyczącymi dostępnej pamięci operacyjnej programu w architekturze 32-bitowej.

4.2 Wykonanie testów jednostkowych i wydajnościowych

Dokładny opis implementacji danych testów jednostkowych oraz wydajnościowych zamieszczony został w rozdziale 6. Poniżej zamieszczono natomiast sposób wykonania danych testów.

W celu sprawdzenia czy wszystkie oferowane funkcje spełniają swoje zadania, istnieje możliwość wykonania wszystkich testów jednostkowych, które zapewniają pełne pokrycie wszystkich implementacji. Aby wykonać testy wystarczy wykonać poniższą komendę z głównego folderu projektu:

```
# Run all unit tests
cd tests/ && make AllTests
```

Natomiast w celu wykonania testów wydajnościowych w pierwszej kolejności wygenerować należy dane testowe, przy czym zalecanym narzędziem jest *dd* dostępne na systemach *Unixo* podobnych. Do repozytorium dodany został skrypt *generateData.sh* znajdujący się w folderze *data/*, który automatycznie wygeneruje potrzebne pliki. Poniżej zamieszczono kolejne komendy pozwalające na wygenerowanie odpowiednich danych oraz wykonanie testów wydajnościowych. Dane testowe nie zostały dołączone do projektu, ponieważ wiele plików posiada znaczne rozmiary, przez co nie mogą być one przetrzymywane w repozytorium.

```
# move to data directory
cd efficiencyTests/data
# Provide executive privileges
chmod +x ./generateData.sh
# Generate data using dd tool or provided script
./generateData.sh
# Move to the efficiencyTests directory and run the tests
cd ../ && make EfficiencyTests
```

5 Opis struktury projektu

Cały projekt podzielony został na cztery zasadnicze foldery:

- documentation
- efficiencyTests
- src
- tests

Poniżej zamieszczono krótki opis każdego z folderów oraz jego zawartości.

5.1 documentation

Folder zawiera sprawozdania w formacie *.tex*, które sukcesywnie oddawane były na kolejnych zajęciach projektowych. Prezentują one cały proces rozwoju naszego projektu oraz pokazują kolejne implementacje oraz modyfikacje algorytmów. W folderze zamieszczono również sprawozdanie z całego projektu.

5.2 efficiencyTests

Folder składa się z plików realizujących testy wydajnościowe zaimplementowanych algorytmów oraz zawiera poprzednie wersje operacji arytmetycznych umieszczone w podfolderze *PastImplementations*, które są znacznie mniej wydajne od funkcji zawartych w bibliotece. Postanowiliśmy pozostawić te implementacje w celu pokazania różnicy czasów wykonywania obu algorytmów w zależności od implementacji. W folderze tym znajduję się również folder *data*, który przechowuje wszystkie potrzebne pliki do przeprowadzenia testów. W tym folderze przeprowadzane jest również profilowanie kodu, którego rezultat znajduje się w *pliku profile-data.txt*.

5.3 src

Folder zawiera główne pliki biblioteki, gdzie umieszczone zostały najbardziej wydajne i optymalne implementacje stworzone w trakcie realizacji projektu. W celu zapewnienia poprawności działania dołączanej biblioteki wszystkie pliki z tego folderu powinny być załączane do danego projektu.

5.4 tests

Folder zawiera wszystkie testy jednostkowe, które podzielone zostały ze względu na sprawdzane treści: *ArithmeticOperationsTests.c* oraz *ParsersTests.c*. W folderze znajduje się również plik *Makefile* w którym zdefiniowano komendy przyśpieszające budowę oraz egzekucję danych testów. Folder *Unity* zawiera kod źródłowy biblioteki dostępnej na licencji MIT, której użyliśmy do realizacji testów jednostkowych.

6 Implementacja

W tym rozdziale zamieszczono dokładny opis zaimplementowanych funkcji wraz z wycinkami kodu, które wymagają komentarza oraz realizują kluczowe funkcje biblioteki. Kod źródłowy również zawiera komentarze, które tłumaczą sposób rozwiązania danego problemu.

Podczas implementacji wszystkich algorytmów staraliśmy się możliwie często wykorzystywać funkcje udostępniane z poziomu języka *C*, aby wykorzystać możliwości optymalizacyjne udostępniane przez kompilator *gcc*. Niestety w wielu przypadkach nie istnieją gotowe moduły w języku *C* pozwalające na bezpośredni dostęp do instrukcji procesora w związku z czym zdefiniowaliśmy własne funkcje w celu zwiększenia wydajności danych algorytmów.

6.1 Sposób reprezentacji liczb

Aby zapewnić dowolną precyzję liczb podczas wykonywanych operacji stworzyliśmy własną strukturę, która pozwala na realizację tego wymagania.

```
typedef struct
{
    unsigned char *number;
    unsigned int numberSize; // bytewise
    int numberPosition;      // bitwise
} TCNumber;
```

Pole *number* jest wskaźnikiem na tablicę charów, która przechowuje liczbę w postaci bajtowej, gdzie liczba reprezentowana jest w systemie U2.

Pole *numberSize* przechowuje długość liczby w reprezentacji bajtowej, ponieważ w języku *C* tablice nie pamiętają swojego rozmiaru.

Natomiast pole *numberPosition* odpowiada za przechowywanie wartości, która określa pozycję najmniej znaczącego bitu liczby.

6.2 Odczyt liczby

Do biblioteki załączyliśmy również dwie funkcje odpowiedzialne za tworzenie nowego obiektu struktury *TCNumber*. Funkcja bez realokacji tablicy znacząco zwiększa wydajność algorytmów i pozwala na zaoszczędzenie znacznych ilości pamięci. Aby odciążyć użytkownika z dealokacji dodaliśmy również funkcję, która poprawnie zwalnia daną liczbę. Wszystkie operacje dealokują argumenty podawane do funkcji aby zaoszczędzić możliwe jak największą ilość pamięci. Dlatego biblioteka posiada dwa rodzaje inicjalizacji obiektów, aby pozwolić na stosowanie operacji zarówno dla tablic znajdujących się na stosie oraz tych alokowanych dynamicznie.

Do wczytywania liczby z standardowego wejścia posłużyliśmy się funkcją *scnaf* z specjalną flagą dostępną w standardzie POSIX, która alokuje wymagany obszar do wczytania liczby. Aby zapewnić również zgodność z systemami nie wspierającymi standardu POSIX dodaliśmy funkcję, która odczytuje liczbę z pliku binarnego.

6.3 Konwersja

Biblioteka oferuje funkcję pozwalającą na konwersję z systemu szesnastkowego, gdzie liczba zapisana jest jako ciąg znaków *ASCII* na system *U2*. Pierwszym etapem konwersji jest ustalenie znaku konwertowanej liczby, wyliczenie jej długości oraz pozycji najmniej znaczącego bitu liczby. Następnie wykorzystujemy funkcję *asciiToByte*, która w zależności od danego znaku szesnastkowego konwertuje go na odpowiednią mu reprezentację bajtową. Następnie posługujemy się przesunięciem bitowym w lewo w celu zachowania poprawnej wartości konwertowanego znaku. Jeśli liczba jest ujemna w ostatnim kroku algorytmu negujemy każdy z bitów a następnie dodajemy do zanegowanej liczby jedynkę, uzyskując w ten sposób reprezentację liczby w systemie *U2*.

6.4 Operacja arytmetyczne

Poniżej znajduje się opis każdego z algorytmów wraz z kawałkami kodu, które obrazują sposób realizacji danego zagadnienia oraz komentarze wyjaśniające daną implementację.

Każda z funkcji napisana została jako odpowiednia metoda w języku *C*, która realizuje wszystkie potrzebne operacje do odpowiedniego przygotowania argumentów oraz wyniku do operacji, natomiast sam algorytm został napisany w języku asembler w celu zwiększenia wydajności oraz wykorzystania instrukcji procesora.

Wszystkie funkcje, które korzystają z mnemoników, zawierają w sobie szereg operacji służących zapewnianiu zgodności z *ABI*, dzięki czemu biblioteka zapewnia pełną integrację wykonywanych operacji z systemem operacyjnym na którym wykonywany jest kod programu oraz pozwala na zaawansowaną kompilację z użyciem *gcc*.

6.4.1 Dodawanie

Pierwszym etapem dodawania jest wyznaczenie rozmiaru wyniku, który w przypadku operacji dodawania jest o jedną pozycję większy od większego z argumentów, ponieważ uwzględnić należy ewentualne przeniesienie na najwyższą pozycję.

```
long long highestPos = addend1->numberPosition + (long long)addend1->numberSize * 8;
if (addend2->numberPosition + (long long)addend2->numberSize * 8 > highestPos)
    highestPos = addend2->numberPosition + (long long)addend2->numberSize * 8;
int lowestPos = addend1->numberPosition;
if (addend2->numberPosition < lowestPos)
    lowestPos = addend2->numberPosition;
unsigned int resultSize = (highestPos - lowestPos) / 8 + 1;
```

Rzutowanie na zmienną typu *long long* jest konieczne, ponieważ dla wielkich liczb pozycja może przyjmować wartości które nie zmieszczą się w zmiennej *int*. Należy pamiętać, że pozycja liczb liczona jest bitowo natomiast wielkość wyniku określana jest w bajtach.

W kolejnym kroku wyliczamy pozycję najbardziej znaczącego bitu drugiej liczby względem pierwszej, ponieważ jedynie pierwsza liczba skalowana jest do rozmiaru wyniku. Wyliczony indeks wykorzystany jest w algorytmie do poprawnej propagacji przeniesienia na pozycje wyższe od pozycji najbardziej znaczącego bitu drugiej liczby.

Następnie skalujemy pierwszy z argumentów do rozmiaru wyniku, ponieważ w nim zapisywany będzie wynik operacji. Operacja skalowania danej liczby do zadanego rozmiaru została dokładnie opisana w rozdziale 6.5.

Później sprawdzamy bit rozszerzenia drugiego z operandów i gdy wynosi on jeden, od pierwszego składnika dodawania odejmujemy jedynkę na pozycji znajdującej się o jedną pozycję wyżej od najbardziej znaczącego bitu drugiej liczby. Operacja ta jest konieczna w celu zapewnienia poprawnej propagacji przeniesienia gdy drugi z operandów jest ujemny.

Następnie odpowiednie zmienne przekazujemy do funkcji *array_ adc*, która wykorzystuje instrukcje procesora do obliczenia wyniku. Należy zauważyć, że instrukcja *cmp* wpływa na flagę przeniesienia, przez co nie może zostać użyta do sprawdzania warunku końca pętli. W celu zachowania poprawnej wartości flagi przeniesienia podczas kolejnych iteracji pętli posłużyliśmy się kombinacją instrukcji *dec* oraz *jns*, które nie powodują zmiany stanu flagi, jednocześnie gwarantując poprawną ilość obiegów pętli. Na końcu zwracamy pierwszy z argumentów, w którym zapisany jest wynik całej operacji.

6.4.2 Odejmowanie

Operacja odejmowania została zrealizowana analogicznie do algorytmu dodawania, z tym że wykorzystana została instrukcja *sbb* oraz pętla służąca do odpowiedniej propagacji pożyczki na pozycjach odjemnej wyższych od pozycji najbardziej znaczącego bitu odjemnika została przedefiniowana aby zapewnić poprawną propagację.

Zmianie uległa też operacja, która wykonywana jest w przypadku ujemnej wartości drugiego operandu. Gdy odjemnik jest ujemny, do odjemnej dodajemy jedynkę na pozycji znajdującej się o jedną pozycję wyżej od najbardziej znaczącego bitu odjemnika, aby poprawnie propagować przeniesienie podczas wyznaczania wyniku.

6.4.3 Mnożenie

Pierwszym etapem mnożenia jest wyznaczenie rozmiaru wyniku, który równy jest sumie rozmiarów mnożnej i mnożnika. Natomiast pozycja najmniej znaczącego bitu iloczynu równa jest sumie pozycji najmniej znaczących bitów mnożnej oraz mnożnika. Podczas operacji mnożenia nie mamy możliwości zapisu wyniku w jednym z operandów, dlatego miejsce na wynik alokowane jest zaraz po wyznaczeniu jego rozmiaru. Do alokacji wykorzystujemy funkcję *calloc*, która zapewnia wyzerowanie zadanego obszaru.

Mnożenie liczb zostało zrealizowane jako funkcja, której parametrami są mnożna oraz zadany bajt mnożnika. Funkcja oblicza kolejne iloczyny częściowe powstałe z pomnożenia danego bajtu mnożnika przez mnożną i sumuje je z wcześniej uzyskanym wynikiem. Funkcja ta wywoływana jest w pętli w której wykonuje się dla każdego bajtu mnożnika dodając uzyskane iloczyny do wyniku, który zaalokowany został przed rozpoczęciem operacji.

6.4.4 Dzielenie

Biblioteka oferuje operacje dzielenia na liczbach przetrzymywanych w wcześniej zdefiniowanej strukturze *TCNumber*, która zrealizowana została zgodnie z algorytmem dzielenia nieodtwarzającego. Cały algorytm zaimplementowany został w języku *C* z wykorzystaniem wcześniej zdefiniowanych operacji dodawania oraz odejmowania. Dzielenie odbywa się poprzeż iteracyjne odejmowanie odpowiednio wyskalowanego dzielnika. Ponieważ rozmiar wyniku jest tylko częściowo definiowany przez dzielną i dzielnik, funkcja wymaga trzeciego parametru, który określa wymaganą precyzję ilorazu.

Pierwszym etapem operacji jest pozbycie się nadmiarowych bitów rozszerzenia dzielnej oraz wyskalowanie jej w taki sposób aby pomieściła wytworzoną resztę oraz poprawnie przechowywała rozszerzenie ilorazu. W tym kroku pozbywamy się również nadmiarowych bitów rozszerzenie dzielnika, aby zminimalizować jego rozmiar, dzięki czemu minimalizujemy liczbę odejmowań w instrukcji *array_ sbb*. Alokujemy miejsce na wynik przy pomocy funkcji *calloc*, dzięki czemu zaalokowana pamięć jest wyzerowana więc podczas wpisywania kolejnych bitów ilorazu rozpatrujemy tylko sytuację gdy wpisać należy 1.

Kolejny krok algorytmu obejmuje sprawdzenie znaku dzielnika i dzielnej, korzystając z operacji bitowych dostępnych w języku *C*. Jeśli znaki są sobie równe to zgodnie z algorytmem dzielenia niedotwarzającego od dzielnej odejmujemy dzielnik, a w przeciwnym wypadku do dzielnej dodajemy drugi z operandów. Podczas realizacji operacji dodawania oraz odejmowania wykorzystujemy wcześniej zaimplementowane funkcje udostępniane przez bibliotekę. Następnie dokonujemy operacji przesunięcia bitowego dzielnika o jedną pozycję w prawo.

Następnie w pętli porównujemy bit rozszerzenia reszty z bitem rozszerzenia dzielnika i gdy są one zgodne do ilorazu dodajemy jeden na ostatniej pozycji a następnie dokonujemy przesunięcia bitowego wyniku w lewo. Podczas wyznaczania wartości warunku sprawdzamy również czy otrzymana reszta nie wynosi zero, w celu zapewnienia poprawnego warunku zatrzymana całej operacji. Operacja sprawdzenia odbywa się poprzez wyznaczenie sumy logicznej wszystkich bitów reszty. Następnie dokonujemy operacji przesunięcia bitowego dzielnika o jedną pozycję w prawo.

6.5 Skalowanie liczby

Biblioteka udostępnia również funkcję pozwalającą na wyskalowanie liczby do zadanego rozmiaru. Ponieważ liczby reprezentowane są w systemie U2, skalowanie odbywa się poprzez uzupełnienie liczby bitami rozszerzenia do zadanej wielkości. Po alokacji miejsca na wyskalowaną liczbę, posłużyliśmy się instrukcją *memcpy*, która znacznie przyśpieszyła proces przepisywania niezmieniających się bajtów, a następnie uzupełniliśmy pozostałe miejsce bitami rozszerzenia.

Ponieważ w wyniku niektórych operacji arytmetycznych, może dojść do sytuacji w której przechowywane będą nadmiorowe bity rozszerzenia lub końcowe bajty zerowe, zaimplementowaliśmy funkcje *scaleUp* oraz *trimExtension*, które pozwalają na minimalizację zajmowanego obszaru przez daną liczbę.

6.6 Testy jednostkowe

Wszystkie funkcje oferowane przez bibliotekę pokryte zostały serią testów, które obejmują większość możliwych przypadków wykorzystania danych metod. Testy zapewniają również poprawności działania oraz pozwalają na szybkie sprawdzenie czy nowo wprowadzone zmiany nie naruszyły wcześniej zaimplementowanych algorytmów.

Do asercji wykorzystaliśmy bibliotekę *Unity*, która oferuje szereg funkcji pozwalających na sprawdzenie poprawności danych operacji. Kluczową asercją jest funkcja pozwalająca na sprawdzenie czy bloki pamięci o zadanym obszarze posiadają równe sobie wartości, dzięki czemu znając z góry wynik danej operacji mogliśmy z łatwością sprawdzić jej poprawność, a w przypadku błędu widzieliśmy, które bajty uległy przekłamaniu.

Poniżej zamieszczono przykładowy test sprawdzający poprawność operacji dodawania dla dwóch liczb dodatnich.

```
void test_add_positive_asm(void)
{
    unsigned char temp1[3] = {0x12, 0x7e, 0x60};
    unsigned char temp2[4] = {0x5d, 0x0a, 0x2b, 0x6e};
    TCNumber *number1 = createTCNumber(temp1, 3, 8);
    TCNumber *number2 = createTCNumber(temp2, 4, -16);
    unsigned char expectedResult[7] = {0x00, 0x12, 0x7e, 0xbd, 0x0a, 0x2b, 0x6e};
    unsigned int expectedSize = 7;
    int expectedPosition = -16;
    TCNumber *result = add_asm(number1, number2);
    TEST_ASSERT_EQUAL_MEMORY(expectedResult, result->number, 7);
    TEST_ASSERT_EQUAL_INT(expectedSize, result->numberSize);
    TEST_ASSERT_EQUAL_INT(expectedPosition, result->numberPosition);
    delete (result);
}
```

Biblioteka wymaga odpowiedniego nazewnictwa funkcji, gdzie każda metoda musi zaczynać się od słowa *test_* . Następnie definiujemy zmienne, które wezmą udział w teście, wykonujemy operację i za pomocą asercji sprawdzamy poprawność wyniku. Dzięki wykorzystaniu asercji na obszarze pamięci na bieżąco otrzymywaliśmy informacje o bajtach, które zawierały nieoprawne wartości a następnie naprawialiśmy dane części algorytmu, które produkowały błędny wynik.

7 Analiza wydajności napisanych algorytmów

W poniższym rozdziale zamieszczono pełną analizę wydajnościową napisanych algorytmów wraz z specyfikacją techniczną maszyny na której wykonywane były dane pomiary.

7.1 Plan eksperymentu

Na każdym z algorytmów przeprowadzono serię testów wydajnościowych, zmieniając rozmiar argumentów oraz dostępne flagi optymalizacyjne podczas kompilacji w celu pokazania zależności pomiędzy rozmiarem operandów a czasem wykonywania oraz możliwości optymalizacyjnych *gcc*.

- Eksperyment został zaimplementowany zgodnie z standardem *ISO C99*.
- Badanie poszczególnych operacji przeprowadzone zostało na losowo generowanych zestawach danych o zadanych wielkościach przy użyciu narzędzia *dd*, gdzie posłużono się partycją */dev/urandom*.
- Wszystkie operacje poza mnożeniem oraz dzieleniem przeprowadzone zostały na operandach o rozmiarach: *100MB, 250MB, 500MB, 750MB, 1GB*. Ze względu na długi czas wykonywania algorytmu mnożenia, drugi argument był stały i wynosił 1MB, natomiast mnożna przyjmowała kolejno wartości: *1KB, 5KB, 10KB, 15KB, 20KB*. W przypadku dzielenia drugi z operandów również był stały i wynosił *20KB*. Ponieważ operacja dzielenia jest kosztowna, dzielna oraz dzielnik zostały dobrane w taki sposób aby zapewnić stosunkowo krótki czas pojedynczej operacji tak aby seria 10 pomiarów wykonywana była w rozsądnym czasie. Jako wielkość dzielnej przyjęto kolejno wartości wynoszące: *5KB, 10KB, 20KB, 50KB, 70KB*.
- Dla operacji dzielenia ilość bitów ilorazu równa była ilości bitów dzielnej.
- Każdy pomiar wykonany został 100 razy z wyjątkiem mnożenia oraz dzielenia dla których liczba prób wynosiła 10 z powodu długiego czasu wykonywania się algorytmu. W sprawozdaniu zamieszczona została średnia uzyskanych pomiarów.
- Do każdej serii pomiarowej zastosowano każdą z dostępnych flag optymalizacyjnych kompilatora *gcc*: *-O0, -O1, -O2, -O3, -Os, -Ofast*.
- Czas wykonywanej operacji zmierzony został za pomocą biblioteki *time.h*.
- Otrzymanym wynikiem jest czas wykonywania danej operacji mierzony w sekundach w zależności od rozmiaru operandów oraz zadanej flagi optymalizacyjnej.
- Podczas przeprowadzania pomiarów wszystkie nieistotne aplikacje zostały zamknięte w celu zminimalizowania błędów pomiarowych oraz wypływu programów trzecich na otrzymany wynik.

7.2 Specyfikacja techniczna wykorzystanych narzędzi

Poniżej zamieszczona została specyfikacja maszyny oraz narzędzi, które wykorzystane zostały do przeprowadzenia wszystkich testów wydajnościowych. Zgodność działania biblioteki testowana była również na systemie Ubuntu 18.04, lecz nie przeprowadzono tam testów wydajnościowych.

System operacyjny	Manjaro 18.0.4 Illyria
Wersja jądra	x86_64 Linux 4.14.113-1-MANJARO
CPU	Intel Core i7-6700HQ @ 8x 3.5GHz
RAM	16 GB - pamięć jednokanałowa
Wersja gcc	8.3.0 (GCC) zgodna z standardem POSIX

Tablica 1: Specyfikacja techniczna

7.3 Wyniki

W rozdziale tym zamieszczono wyniki przeprowadzonych testów wraz z wykresami, które prezentują zależność czasową wykonywania algorytmu od rozmiaru operandów oraz flagi optymalizacyjnej zastosowanej w trakcie kompilacji. Podczas interpretacji wyników należy uwzględnić ewentualne niepewności pomiarowe, które spowodowane są innymi procesami, których nie można było zakończyć przed wykonaniem pomiarów.

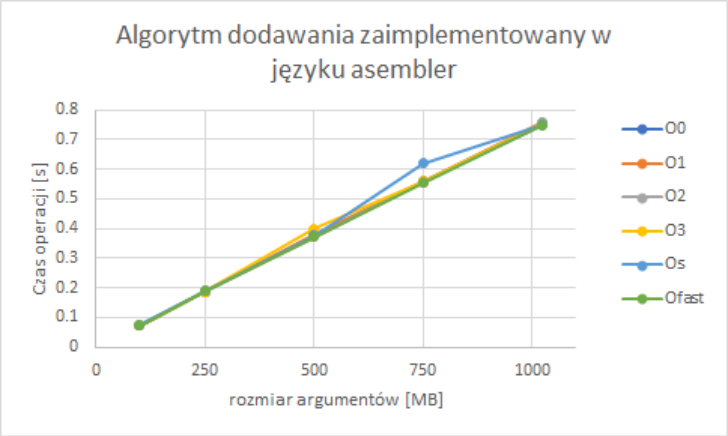
Wszystkie dane zostały przedstawione zarówno w postaci tabelarycznej jak i na wykresach w celu klarownego zobrazowania danych oraz zależności między danymi rozmiarami operandów oraz czasem wykonywania w zależności od zadanej flagi optymalizacyjnej.

7.4 Rezultat badań poszczególnych implementacji

Badanie poszczególnych implementacji podzielone zostało na sekcje ze względu na analizowane algorytmy.

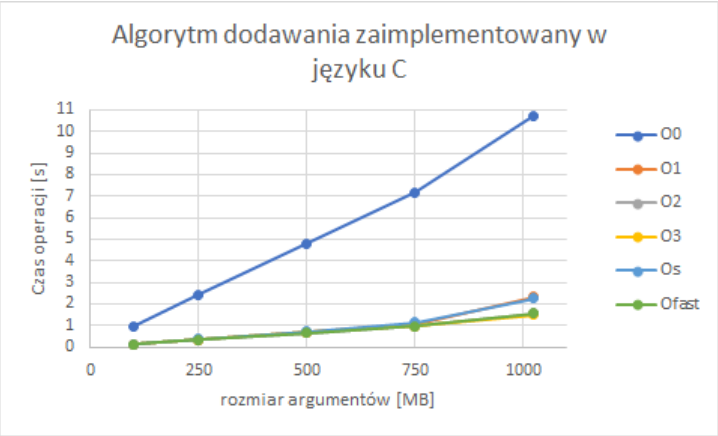
7.4.1 Operacja dodawania

Algorytm dodawania zaimplementowany w języku asembler						
rozmiar [MB]	Czas operacji dla danej flagi optymalizacyjnej [s]					
[100 - 1024]	O0	O1	O2	O3	Os	Ofast
100	0.074141	0.073895	0.073681	0.073571	0.07441	0.074169
250	0.186428	0.189979	0.187065	0.187214	0.189474	0.188714
500	0.375807	0.382607	0.370654	0.399656	0.377289	0.372111
750	0.559446	0.555589	0.558106	0.558585	0.620842	0.553397
1024	0.7526	0.758936	0.756261	0.75337	0.752351	0.750059



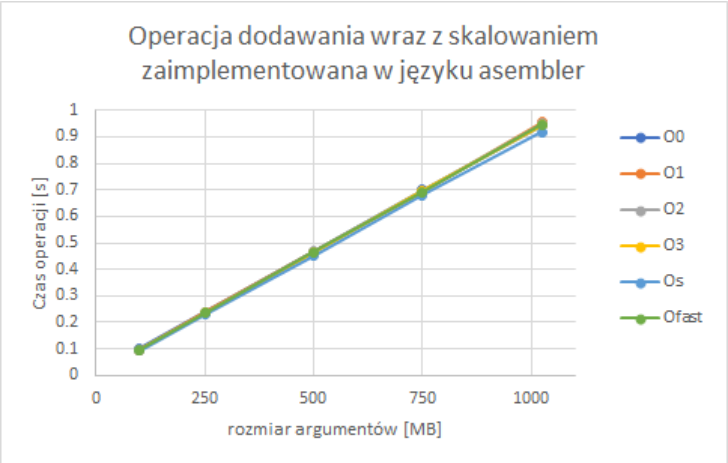
Powyższy wykres prezentuje zależność czasową dla funkcji zaimplementowanej w języku asembler, która realizuje dodanie dwóch zadanych liczb w postaci tablicy bajtów z wykorzystaniem instrukcji *adc*. Powyższe badanie miało na celu sprawdzenie algorytmu bez procesu skalowania argumentów. Otrzymane wynik potwierdzają teorię, która głosi, że operacja dodawania zależna jest od wielkości operandów i wraz ze wzrostem rozmiarów składników czas operacji rośnie. Należy również zauważyć, że zastosowanie różnych flag optymalizacyjnych w niewielkim stopniu wpływa na czas wykonania algorytmu. Spowodowane jest to implementacją całej funkcji w języku asemblera, co powoduje brak możliwości optymalizacji kodu przez *gcc*. Stosunkowy długi czas wykonywania operacji spowodowany jest również implementacją algorytmu, gdzie w każdej iteracji dodawany jest bajt po bajcie. Pierwszym krokiem jaki należałoby podjąć aby zoptymalizować algorytm jest dodawanie ośmiu bajtów w każdej iteracji, dzięki czemu osiągnęlibyśmy przynajmniej ośmiokrotnie lepszy rezultat.

Algorytm dodawania zaimplementowany w języku C						
rozmiar [MB]	Czas operacji dla danej flagi optymalizacyjnej [s]					
[100 - 1024]	O0	O1	O2	O3	Os	Ofast
100	0.959596	0.137802	0.132386	0.131649	0.144522	0.13377
250	2.434677	0.346439	0.322735	0.325991	0.357946	0.327882
500	4.813448	0.690317	0.649277	0.677997	0.719295	0.655606
750	7.163609	1.025286	0.975917	0.978341	1.12465	0.975454
1024	10.743684	2.313034	1.567546	1.498769	2.265664	1.546174



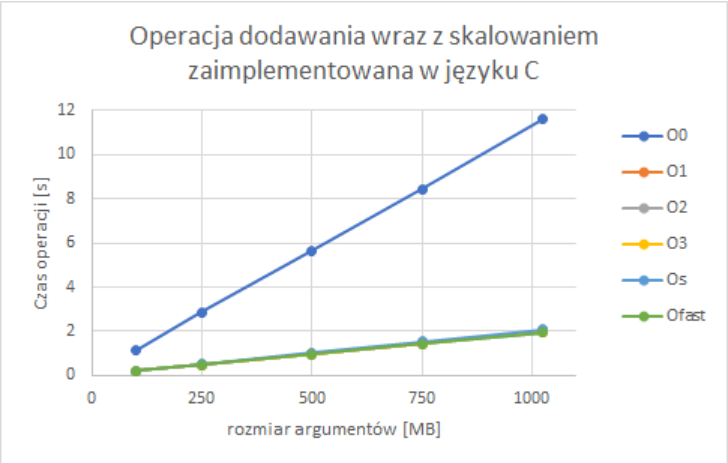
Analizę naiwnej implementacji w języku *C* przeprowadziliśmy w celu pokazania różnicy pomiędzy czasem wykonywania algorytmu wykorzystującego instrukcje asemblera a tym zdefiniowanym w języku *C*. Dane przedstawione w formie tabelarycznej pokazują znaczącą różnicę pomiędzy czasami wykonania operacji. Ponieważ całość algorytmu zdefiniowana jest w języku *C*, na wykresie widzimy znacząco skrócony czas wykonywania dla flag odpowiadających wyższym stopniom optymalizacji. Zauważyć należy również, że pomimo znaczącej różnicy w czasie pomiędzy implementacją w języku *C* a asemblerem, kolejne czasy dla większych operandów nie odbiegają od siebie tak bardzo jak w przypadku pierwszego algorytmu. Główną wadą tego rozwiązania jest brak wykorzystania instrukcji *adc*, która znacznie przyśpiesza proces dodawania i rozwiązuje problem propagacji przeniesienia na dalsze pozycje.

Operacja dodawania wraz z skalowaniem argumentu zaimplementowana w języku asembler						
rozmiar [MB]	Czas operacji dla danej flagi optymalizacyjnej [s]					
[100 - 1024]	O0	O1	O2	O3	Os	Ofast
100	0.103505	0.097359	0.097064	0.097633	0.095075	0.09779
250	0.238054	0.238679	0.234887	0.234815	0.229957	0.233979
500	0.464266	0.465176	0.467217	0.460531	0.450481	0.464453
750	0.699358	0.691717	0.696312	0.69819	0.679542	0.692367
1024	0.943754	0.954195	0.950186	0.944086	0.916669	0.946757



Wykres nie odbiega znacznie od tego prezentującego zależność dla samego algorytmu dodawania. Ponieważ tylko jeden z argumentów musi być skalowany do rozmiaru wyniku, wykorzystując funkcję *memcpy* znacznie zmniejszyliśmy czas wykonania operacji skalowania, dzięki czemu narzut czasu jaki nakłada ona na sam algorytm wynosi około *0.1 s*. Operacja skalowania napisana została w całości w języku *C*, lecz jak widać na wykresie czas wykonania całej operacji w małym stopniu zależy od zastosowanej flagi optymalizacyjnej.

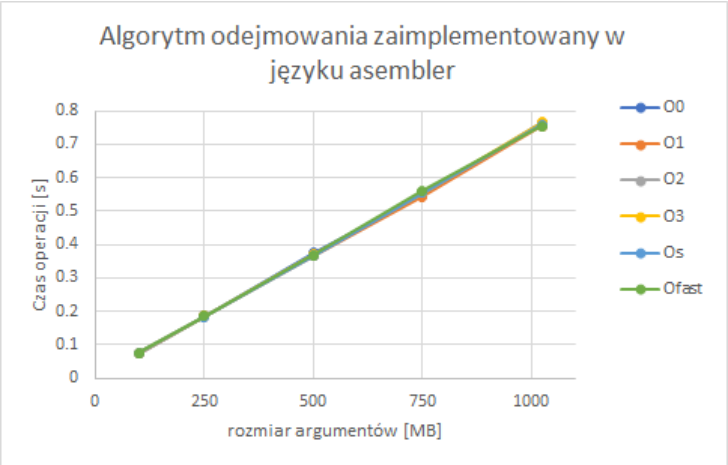
Operacja dodawania wraz z skalowaniem argumentu zaimplementowana w języku C						
rozmiar [MB]	Czas operacji dla danej flagi optymalizacyjnej [s]					
[100 - 1024]	O0	O1	O2	O3	Os	Ofast
100	1.141805	0.2024	0.196861	0.197863	0.212856	0.198667
250	2.851668	0.494804	0.48209	0.482317	0.513929	0.481932
500	5.644676	0.99328	0.959344	0.963643	1.030213	0.954745
750	8.43348	1.466882	1.446451	1.429362	1.540946	1.442489
1024	11.58157	2.014692	1.955252	1.961645	2.071897	1.959199



Powyższy diagram pokazuje zależność czasową algorytmu dodawania wraz z narzutem funkcji skalującej pierwszy z składników do rozmiaru wyniku. Cały algorytm napisany został w języku *C*, dzięki czemu zaobserwować możemy znacznie krótszy czas wykonania dla wyższych flag optymalizacyjnych. Pomimo większej optymalizacji zapewnionej przez kompilator *gcc* czas wykonania całej operacji jest znacząco dłuższy od algorytmu zaimplementowanego z użyciem instrukcji *adc*. Tak jak wcześniej narzut funkcji odpowiedzialnej za skalowanie jest niewielki w porównaniu do czasu potrzebnego do obliczenia wyniku.

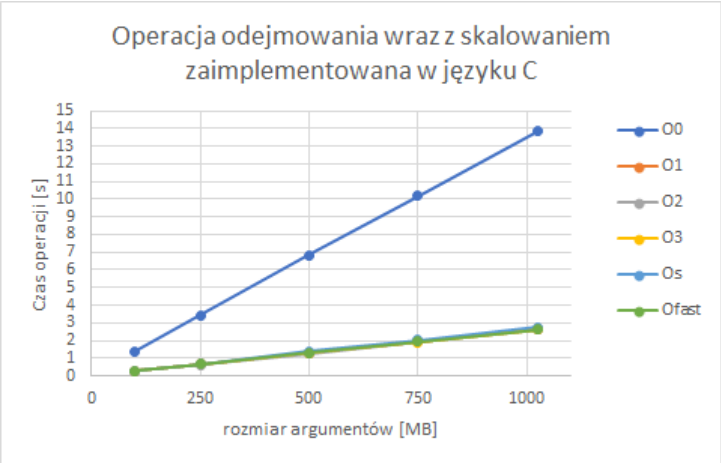
7.4.2 Operacja odejmowania

Algorytm odejmowania zaimplementowany w języku asembler						
rozmiar [MB]	Czas operacji dla danej flagi optymalizacyjnej [s]					
[100 - 1024]	O0	O1	O2	O3	Os	Ofast
100	0.074688	0.07445	0.074733	0.07577	0.076039	0.075567
250	0.187221	0.187265	0.183571	0.186805	0.184681	0.186967
500	0.373943	0.365276	0.368545	0.370165	0.366733	0.368875
750	0.550903	0.543541	0.554511	0.554136	0.55541	0.560674
1024	0.763826	0.753965	0.761476	0.766865	0.759583	0.75566



Algorytm odejmowania został zaimplementowany analogicznie do instrukcji wykorzystanych w operacji dodawania z tym, że zamiast instrukcji *adc* wykorzystano mnemonik *sbb*. Potwierdzają to również dane w tabeli oraz wykres, gdzie wyniki zbliżone są do tych otrzymanych podczas badania algorytmu dodawania. Całość algorytmu zrealizowana została w języku asemblera w związku z czym optymalizacja zapewniana przez kompilator *gcc* nie poprawia znacznie czasów operacji.

Operacja odejmowania wraz z skalowaniem argumentu zaimplementowana w języku C						
rozmiar [MB]	Czas operacji dla danej flagi optymalizacyjnej [s]					
[100 - 1024]	O0	O1	O2	O3	Os	Ofast
100	1.372728	0.268066	0.2614	0.262618	0.278764	0.264638
250	3.43102	0.665387	0.643401	0.644065	0.675299	0.649144
500	6.814707	1.317293	1.280693	1.287824	1.373065	1.293463
750	10.162231	1.951098	1.930281	1.916266	2.027647	1.923487
1024	13.850198	2.675922	2.620927	2.616287	2.732531	2.610438

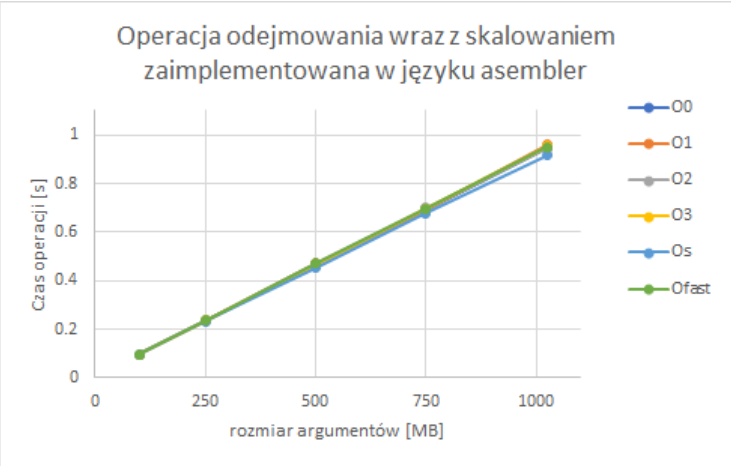


Operacja odejmowania w języku C zrealizowana została z wykorzystaniem zależności:

$$\bar{X} - \bar{Y} = X + \underline{Y} = X + \bar{Y} + ulp \tag{1}$$

Algorytm polegał na zanegowaniu każdego z bitów drugiego z operandów a następnie dodanie jedynki na najmniej znaczącej pozycji. Kolejnym krokiem było wykorzystanie naiwnej implementacji dodawania w języku C na odjemnej oraz wcześniej wyznaczonym odjemniku. Sposób ten był bardzo nieefektywny, ponieważ W pierwszym kroku należało przejrzeć całą tablicę i zanegować każdą pozycję w tablicy, która przechowuje drugi składnik odejmowania. Następnie dodawano jedynkę na najmniej znaczącej pozycji, bez wykorzystania instrukcji *adc*, przez co analogicznie należało przejrzeć całą tablicę w celu zapewnienia poprawnej propagacji przeniesienia. Rezultat wykonania wszystkich zdefiniowanych wyżej operacji, służących do poprawnego obliczenia wyniku widoczny jest na wykresie oraz w tabeli, gdzie czas jest znacząco dłuższy od czasu wykonania sumowania. Algorytm ten jest znacząco mniej wydajny od tego zaimplementowanego z użyciem instrukcji *sbb*, nawet po uwzględnieniu optymalizacji zapewnianej przez *gcc*.

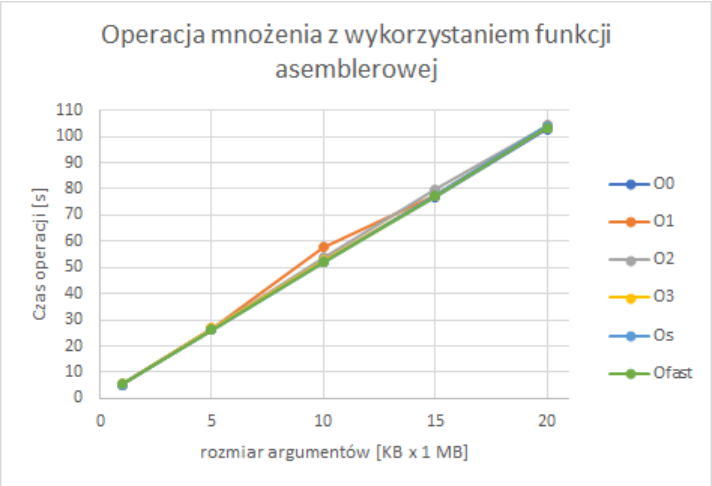
Operacja odejmowania wraz z skalowaniem argumentu zaimplementowana w języku asembler						
rozmiar [MB]	Czas operacji dla danej flagi optymalizacyjnej [s]					
[100 - 1024]	O0	O1	O2	O3	Os	Ofast
100	0.096792	0.096728	0.097034	0.097697	0.095425	0.098064
250	0.237374	0.236262	0.234378	0.234601	0.231697	0.235693
500	0.462814	0.470034	0.467043	0.462492	0.451631	0.469297
750	0.692296	0.69468	0.699802	0.695344	0.679241	0.697017
1024	0.945461	0.957504	0.943473	0.953357	0.914189	0.949194



Tak jak w przypadku algorytmu dodawania, proces skalowania argumentu do rozmiaru wyniku zapewnia narzut o średniej wartości *0.2 s* na czas wykonywania operacji. W porównaniu do algorytmu zaimplementowanego w języku *C* wypada on znacznie lepiej, pomimo znikomej optymalizacji ze strony kompilatora. Dane zaprezentowane na wykresie oraz w tabeli potwierdzają analogiczne implementację algorytmu odejmowania oraz dodawania, ponieważ poszczególne czasy wykonania dla operandów o równych wielkościach nie odbiegają od siebie znacząco.

7.4.3 Operacja mnożenia

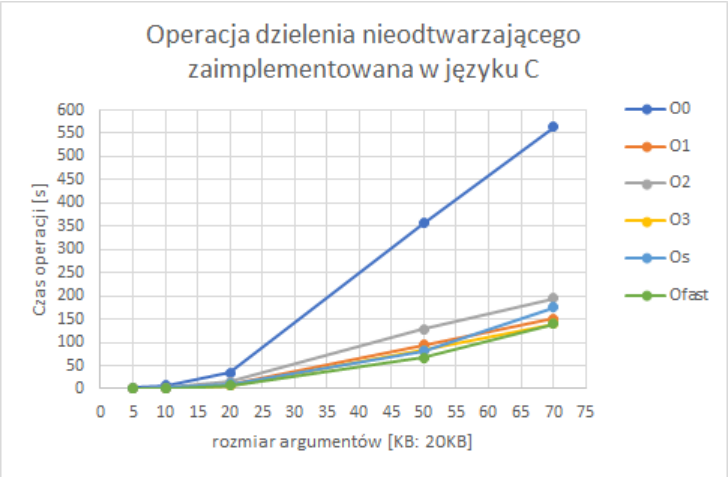
Operacja mnożenia zaimplementowana z wykorzystaniem funckji zdefiniowanej w asemblerze						
rozmiar	Czas operacji dla danej flagi optymalizacyjnej [s]					
[KB-MB]	O0	O1	O2	O3	Os	Ofast
1-1	5.283762	5.325324	5.334093	5.416204	5.271519	5.48234
5-1	26.605809	26.837862	26.381239	26.735731	26.234717	26.242277
10-1	52.274452	57.747996	53.694794	52.51533	51.978892	52.138293
15-1	76.979769	77.853978	79.687914	77.603236	77.443207	77.296892
20-1	102.684943	104.062236	104.04851	103.218145	103.915413	103.358728



Mnożenie znacznie przekracza czasy wykonywania wszystkich poprzednich implementacji, ponieważ jest operacją, która wymaga największego nakładu pracy w celu wyznaczenia całego iloczynu. Aby uzyskać ostateczny wynik wyznaczyć należy wszystkie iloczyny częściowe a następnie odpowiednio je posumować, co przekłada się na czasy widoczne w tabeli oraz na wykresie. Aby wykorzystać instrukcję *mul*, zdefiniowaliśmy funkcję w języku asemblera, która odpowiedzialna jest za przemnożenie kolejnego bajta mnożnika przez całą mnożną oraz odpowiednie posumowanie uzyskanych iloczynów z obecnym iloczynem wynikowym. Skutkuje to słabą optymalizacją ze strony *gcc*, lecz zapewnia większą wydajność niż algorytm zaimplementowany w czystym języku *C*. Mnożna oraz mnożnik zostały odpowiednio dobrane tak aby zapewnić sensowny czas wykonania operacji oraz pokazać zależność pomiędzy kolejnymi czasami egzekucji algorytmu oraz wielkością operandów.

7.4.4 Operacja dzielenia

Operacja dzielenia nieodtwarzającego zaimplementowana w języku C						
rozmiar	Czas operacji dla danej flagi optymalizacyjnej [s]					
[KB-KB]	O0	O1	O2	O3	Os	Ofast
5:20	1.663121	0.398279	0.671055	0.368187	0.46069	0.35623
10:20	6.696112	1.57283	2.689526	1.48084	1.820485	1.477208
20:20	35.079256	9.385996	15.223272	7.932796	9.810787	7.921247
50:20	356.12477	94.605824	128.735818	82.468434	81.457742	66.546889
70:20	563.269941	150.554117	194.808472	140.31981	175.523364	139.570556



Aby zapewnić odpowiednią skalowalność algorytmu, cała operacja zrealizowana została jako iteracyjne odejmowanie odpowiednio wyskalowanego dzielnika. Całość algorytmu napisana została w języku *C*, dzięki czemu wraz z kompilacją z wyższymi stopniami optymalizacji zauważyć możemy, istotnie zmniejszający się czas obliczenia ilorazu. Pomimo braku wykorzystania instrukcji *div*, algorytm poprawnie realizuje swoje zadanie, oraz osiąga względnie akceptowalny czas wykonania dla małych operandów. Porównując otrzymane wynik z rezultatem badań poprzednich algorytmów, zauważyć można, że zaproponowana przez nas implementacja dzielenie nie charakteryzuje się liniową zależnością tak jak wcześniejsze operacje. Spowodowane jest to dużym narzutem czasowym wprowadzonym przez operację skalowania dzielnika podczas kolejnych iteracji algorytmu oraz wykonanie dużej liczby odejmowań w celu wyznaczenie ilorazu. Algorytm bazuje również na wcześniej zaimplementowanych operacjach dodawania i odejmowania, w związku z czym zaproponowana poprawa wcześniej wymienionych algorytmów wpłynęłaby również na poprawę wydajności dzielenia.

8 Profilowanie oraz analiza pamięciowa

W celu dokładnej analizy zaimplementowanych algorytmów posłużono się narzędziem *Gprof*, które umożliwia profilowanie wykonywanego kodu programu. Wynikiem wykorzystania narzędzia jest raport w którym pokazane jest jaka część wykonywanych instrukcji była najbardziej czasochłonna oraz które z nich były najczęściej wykonywane. Następnie wykorzystano dostępne flagi kompilatora: *-fsanitize* oraz *-fno-omit-frame-pointer*, dzięki którym zlokalizowano miejsca w programie w których zabrakło deaalokacji wcześniej zajętych obszarów. Ostatecznie plik wykonywalny, który testował wszystkie funkcje oferowane przez bibliotekę został poddany analizie z użyciem narzędzia *valgrind* aby przeprowadzić dokładną analizę pamięciową oraz zorientować się jak wiele pamięci wykorzystuje program podczas swojego działania.

8.1 Profilowanie przy użyciu narzędzia *Gprof*

Poniżej zamieszczono przykładowy kod programu, testujący wszystkie funkcje dostarczane przez bibliotekę, gdzie operandami danych metod są liczby o największych rozmiarach użytych podczas badania algorytmów. Wielkość argumentów jest kluczowa, ponieważ znacząco wpływa na czas egzekucji programu, dzięki czemu podczas profilowania możemy zaobserwować najbardziej czasochłonne operacje. Profilowanie przeprowadzone zostało bez optymalizacji dostarczanej przez kompilator *gcc* w celu uwidocznienia różnicy w czasie wykonywania danych algorytmów.

```
#include "EfficiencyTests.h"

int main()
{
    TCNumber *firstNumber = getNumberFromBinaryFile(firstNumber1GBPath, 0);
    TCNumber *secondNumber = getNumberFromBinaryFile(secondNumber1GBPath, 0);
    TCNumber *result;

    // Run add operation
    result = add(firstNumber, secondNumber);
    delete (result);

    // Recreate structures
    firstNumber = getNumberFromBinaryFile(firstNumber1GBPath, 0);
    secondNumber = getNumberFromBinaryFile(secondNumber1GBPath, 0);

    // Run subtraction operation
    result = subtract(firstNumber, secondNumber);
    delete (result);

    // Files for multiplication
    firstNumber = getNumberFromBinaryFile(firstNumber1MBPath, 0);
    secondNumber = getNumberFromBinaryFile(firstNumber20KBPath, 0);

    // Test multiplication algorithm
    result = multiply(firstNumber, secondNumber);
    delete (result);

    // Files for division
    firstNumber = getNumberFromBinaryFile(secondNumber50KBPath, 0);
    secondNumber = getNumberFromBinaryFile(firstNumber20KBPath, 0);

    result = divide(firstNumber, secondNumber, 0);
    delete (result);

    return 0;
}
```

Poniższa tabela przedstawia wynik profilowania powyższego programu wraz z wszystkimi informacja dostarczonymi przez narzędzie *Gprof*. Tabela prezentuje rozkład czasu ze względu na wykonywania funkcje oraz ilość ich wywołań.

Profilowanie z wykorzystaniem narzędzia Gprof						
% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
31.35	263.91	263.91	655354.00	0.00	0.00	array_shift_right
28.67	505.26	241.35	655352.00	0.00	0.00	array_shift_left
17.20	650.05	144.79	1.00	144.79	650.05	divide
8.64	722.75	72.70				ptmul
5.52	769.22	46.47				mulcarry
4.32	805.61	39.39				diff
2.14	823.63	18.02				sum
2.05	840.90	17.27				mulcarryout
0.10	841.70	0.80				ptprod
0.00	841.70	0.00	14.00	0.00	0.00	createTCNumber_no_realloc
0.00	841.70	0.00	14.00	0.00	0.00	delete
0.00	841.70	0.00	8.00	0.00	0.00	getNumberFromBinaryFIle
0.00	841.70	0.00	3.00	0.00	0.00	scaleNumber
0.00	841.70	0.00	2.00	0.00	0.00	trimExtension
0.00	841.70	0.00	1.00	0.00	0.00	add
0.00	841.70	0.00	1.00	0.00	0.00	multiply
0.00	841.70	0.00	1.00	0.00	0.00	subtract

- Kolumna *% time* prezentuje ilość czasu wyrażoną w procentach w stosunku do całego czasu wykonywania programu.
- *Cumulative seconds* zawiera ilość sekund przeznaczonych na wykonanie wszystkich funkcji, które znajdują się na daną metodą w tabeli wraz z czasem wykonania funkcji w danym wierszu.
- *Self seconds* pokazuje czas wykonywania konkretnej funkcji wyrażony w sekundach.
- Kolumna *Calls* oznacza ilość wywołań danej funkcji.
- *Self s/call* prezentuje średnią ilość sekund wykonania danej metody przypadającej na jedne wywołanie funkcji.
- *Total s/call* zawiera średni czas wykonania funkcji wraz z wszystkimi metodami, które dana funkcja wykonuje z swojego ciała.
- *Name* zawiera nazwę funkcji, której dotyczą dane w danym wierszu.

Analizując powyższą tabelę, możemy wyciągnąć wnioski na temat najbardziej czasochłonnych procesów, które odbywają się podczas egzekucji programu dzięki czemu otrzymujemy informacje, które algorytmy wymagają optymalizacji. W powyższej tabeli widzimy, że operacja dzielenie jest zdecydowanie najbardziej czasochłonna, ponieważ zrealizowana została jako iteracyjne odejmowanie wyskalowanego dzielnika oraz odpowiednie przesuwanie rezultatu, które zajmuje zdecydowaną większość czasu. Operacja mnożenia wykorzystuje instrukcję *mul*, dzięki czemu osiąga lepszy czas wykonywania całego algorytmu w stosunku do algorytmu dzielenia. Etykiety *ptmul, mulcarry, diff, sum, mullcarryout* oraz *ptprod* definiują miejsca w kodzie asemblerowym, gdzie realizowane są kluczowe operacji potrzebne do wykonania danego algorytmu. Natomiast funkcje *add, multiply, subtract* są jedynie metodami zdefiniowanymi w języku *C*, które odpowiadają za przygotowanie argumentów do algorytmu zdefiniowanego w języku asemblera w związku z czym zajmują znikome ilości czasu.

Poniżej zamieszczono drzewo wywołań programu, które posortowane zostało ze względu na czas wywołania każdej z funkcji.

```
index % time    self  children    called         name
                144.79  505.26       1/1           main [2]
[1]     77.2  144.79  505.26       1             divide [1]
                263.91    0.00  655354/655354      array_shift_right [3]
                241.35    0.00  655352/655352      array_shift_left [4]
                  0.00    0.00       3/14          delete [12]
                  0.00    0.00       2/2           trimExtension [15]
                  0.00    0.00       2/14          createTCNumber_no_realloc [11]
                  0.00    0.00       1/3           scaleNumber [14]
-----------------------------------------------
[2]     77.2    0.00  650.05                       main [2]
                144.79  505.26       1/1           divide [1]
                  0.00    0.00       8/8           getNumberFromBinaryFile [13]
                  0.00    0.00       4/14          delete [12]
                  0.00    0.00       1/1           add [16]
                  0.00    0.00       1/1           subtract [18]
                  0.00    0.00       1/1           multiply [17]
-----------------------------------------------
                263.91    0.00  655354/655354      divide [1]
[3]     31.4  263.91    0.00  655354              array_shift_right [3]
-----------------------------------------------
                241.35    0.00  655352/655352      divide [1]
[4]     28.7  241.35    0.00  655352              array_shift_left [4]
-----------------------------------------------
[5]      8.6   72.70    0.00                       ptmul [5]
-----------------------------------------------
[6]      5.5   46.47    0.00                       mulcarry [6]
-----------------------------------------------
[7]      4.3   36.39    0.00                       diff [7]
-----------------------------------------------
[8]      2.1   18.02    0.00                       sum [8]
-----------------------------------------------
[9]      2.1   17.27    0.00                       mulcarryout [9]
-----------------------------------------------
[10]     0.1    0.80    0.00                       ptprod [10]
-----------------------------------------------
                  0.00    0.00       1/14          multiply [17]
                  0.00    0.00       2/14          divide [1]
                  0.00    0.00       3/14          scaleNumber [14]
                  0.00    0.00       8/14          getNumberFromBinaryFile [13]
[11]     0.0    0.00    0.00       14              createTCNumber_no_realloc [11]
-----------------------------------------------
                  0.00    0.00       1/14          add [16]
                  0.00    0.00       1/14          subtract [18]
                  0.00    0.00       2/14          multiply [17]
                  0.00    0.00       3/14          divide [1]
                  0.00    0.00       3/14          scaleNumber [14]
                  0.00    0.00       4/14          main [2]
[12]     0.0    0.00    0.00       14              delete [12]
-----------------------------------------------
                  0.00    0.00       8/8           main [2]
[13]     0.0    0.00    0.00       8               getNumberFromBinaryFile [13]
                  0.00    0.00       8/14          createTCNumber_no_realloc [11]
-----------------------------------------------
                  0.00    0.00       1/3           add [16]
                  0.00    0.00       1/3           subtract [18]
                  0.00    0.00       1/3           divide [1]
[14]     0.0    0.00    0.00       3               scaleNumber [14]
                  0.00    0.00       3/14          delete [12]
                  0.00    0.00       3/14          createTCNumber_no_realloc [11]
-----------------------------------------------
                  0.00    0.00       2/2           divide [1]
[15]     0.0    0.00    0.00       2               trimExtension [15]
-----------------------------------------------
                  0.00    0.00       1/1           main [2]
[16]     0.0    0.00    0.00       1               add [16]
                  0.00    0.00       1/3           scaleNumber [14]
                  0.00    0.00       1/14          delete [12]
-----------------------------------------------
                  0.00    0.00       1/1           main [2]
[17]     0.0    0.00    0.00       1               multiply [17]
                  0.00    0.00       2/14          delete [12]
                  0.00    0.00       1/14          createTCNumber_no_realloc [11]
-----------------------------------------------
                  0.00    0.00       1/1           main [2]
[18]     0.0    0.00    0.00       1               subtract [18]
                  0.00    0.00       1/3           scaleNumber [14]
                  0.00    0.00       1/14          delete [12]
-----------------------------------------------
```

Drzewo również obrazuje wnioski, które zaprezentowane zostały w poprzednim akapicie. Krokami, które podjąć należy w celu optymalizacji działania biblioteki jest próba zmniejszenia liczby wywołań funkcji *array_shift_right* oraz *array_shift_left*, które zajmują zdecydowaną większość czasu działania programu. Podczas analizy algorytmów pod każdym z wykresów zamieszczono sugestie, które wykorzystać można podczas optymalizacji danych algorytmów, co wpłynęłoby również na wyniki kolejnego profilowania. Statyczna analiza kodu pozwala na dokładną analizę czasową wykonywanych algorytmów oraz dostarcza przydatnych informacji na temat ilości wywołań danych funkcji.

8.2 Wykorzystanie narzędzia *Valgrind* oraz flagi *-fsanitize* do detekcji wycieków pamięci

Flagi *-fsanitize=address* oraz *-fno-omit-frame-pointer* okazały się kluczowe podczas implementacji algorytmu dzielenia, ponieważ wymaga on dużej liczby operacji na pamięci, a zastosowane flagi pozwoliły na dynamiczną analizę wykonywanego kodu, dzięki czemu na bieżąco lokalizowaliśmy błędne adresowanie oraz wycieki pamięci. Flagi pozwalające na dynamiczną analizę kodu w połączeniu z flagą *-g*, prezentowały dokładną linię kodu w której występował dany błąd, dzięki czemu szybko lokalizowaliśmy naruszenie pamięci.

Następnie po pozbyciu się wszystkich problemów związanych z zarządzaniem pamięcią wykorzystaliśmy program *Valgrind*, który prezentował dokładniejsze informacje dotyczące zarządzania pamięcią.

Poniżej zaprezentowano wynik działania programu wraz z jego podsumowaniem dostarczonym przez analizator.

```
==13036== Memcheck, a memory error detector
==13036== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13036== Using Valgrind-3.14.0-3a3000290b-20181009X and LibVEX;
==13036== Command: ./oiakProject.exe
==13036== Parent PID: 12008
==13036==
--13036--
--13036-- Valgrind options:
--13036--    --leak-check=full
--13036--    -v
--13036--    --log-file=valgrind.txt
--13036-- Contents of /proc/version:
--13036--   Linux version 4.14.113-1-MANJARO (builduser@lancaster) (gcc version 8.3.0
(GCC)) #1 SMP PREEMPT Sun Apr 21 12:03:01 UTC 2019
--13036--
--13036-- Arch and hwcaps: X86, LittleEndian, x86-mmxext-ssel-sse2-sse3-lzcnt
--13036-- Page sizes: currently 4096, max supported 4096
--13036-- Valgrind library directory: /usr/lib/valgrind
==13036==
==13036== HEAP SUMMARY:
==13036==     in use at exit: 0 bytes in 0 blocks
==13036==   total heap usage: 35 allocs, 35 frees, 1,331 bytes allocated
==13036==
==13036== All heap blocks were freed -- no leaks are possible
==13036==
==13036== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==13036== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Analiza pamięciowa przeprowadzona została na programie, który wykorzystywał wszystkie funkcje oferowane przez bibliotekę, dzięki czemu mamy pewność, że wykonywany kod nie zawiera miejsc w których dochodzi do wycieku pamięci. Dynamiczny analizator kodu pozwolił wykryć błędy w alokacji, pokazał błędy podczas wykorzystania funkcji *memcpy* oraz pomagał podczas debugowania kodu, który wykonywał wiele operacji na stercie.

9 Wnioski reasumujące cały projekt

Główne cele projektu zostały zrealizowane zgodnie z założeniami zdefiniowanymi na początku semestru. Biblioteka napisana w języku *C* oferuje cztery podstawowe operacje arytmetyczne, operujące na liczbach dowolnej precyzji.

Wybór wewnętrznej reprezentacji *U2*, był bardzo trafny, ponieważ reprezentacja ta jest używana przez język asemblera, dzięki czemu nie musieliśmy przeprowadzać dodatkowych operacji w celu wykorzystania ich podczas operacji na mnemonikach.

W projekcie silnie polegaliśmy oraz wykorzystywaliśmy łatwość łączenia kodu języka *C* z kodem asemblerowym, aby możliwe najlepiej zoptymalizować dane algorytmy. Implementacja głównych algorytmów w języku asemblera przełożyła się na optymalizację zapewnianą przez *gcc*, co widoczne jest na wykresach, gdzie flagi optymalizacyjne dla funkcji napisanych w asemblerze nie odgrywają wielkiej roli. Pomimo to algorytmy te znacznie przewyższają wydajnością naiwne implementacje w języku *C*, które napisaliśmy na początku zajęć.

Kluczowym elementem projektu było wykorzystanie w projekcie biblioteki *Unity*, która pozwoliła nam na pełne pokrycie testami każdej z operacji oraz zapewniała poprawność danych operacji podczas ewentualnej zmiany algorytmu. Testy jednostkowe pozwoliły nam na bieżąco sprawdzać poprawność danych algorytmów„ dzięki czemu szybciej dochodziliśmy do rozwiązania problemu.

Ostatni etap dotyczył aspektu badawczego zaimplementowanych algorytmów. Ponieważ podczas pierwszych prób implementacji nie do końca zrozumieliśmy zamysł wykorzystania instrukcji asemblerowych, które służyć miały poprawie wydajności algorytmów, stworzyliśmy funkcje, które poprawnie realizowały dane zadanie lecz robiły to w sposób niewydajny. Naiwne implementacje odłączyliśmy od głównego projektu, ale wykorzystaliśmy je podczas przeprowadzania badań w celu pokazania różnicy pomiędzy danymi algorytmami. Cały przebieg eksperymentu został udokumentowany w sprawozdaniu, poprzez wykresy oraz tabele z danymi wraz z płynącymi z tych danych wnioskami.

Realizowany temat dobrze nakreślił nam problem reprezentacji liczb w pamięci komputera oraz wyzwania dotyczące zapewnienia dowolnej precyzji wykonywanych operacji. Istotnym elementem projektu był sposób reprezentacji liczb, ponieważ definiował on później wykonywane operacje. Wykorzystując strukturę do reprezentacji liczby, która przechowuje rozmiar liczby, daną liczbę w systemie U2, oraz pozycję najmniej znaczącego bitu, rozwiązaliśmy problem odpowiedniego skalowania liczby oraz odpowiedniego wyznaczania jej pozycji podczas wykonywania danych operacji.

Literatura

[1] Materiały dostępne na stronie zakładu architektury komputerów
`http://zak.ict.pwr.wroc.pl/`

[2] ThrowTheSwitch, Unity
`http://www.throwtheswitch.org/unity`
`https://github.com/ThrowTheSwitch/Unity`

[3] Valgrind
`http://www.valgrind.org/`

[4] Travis CI
`https://docs.travis-ci.com`