# Organizacja i Architektura Komputerów - Projekt

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


| | |
|---|---|
| *Uczelnia i wydział:* | **Politechnika Wrocławska**<br>**Wydział Elektroniki** |
| *Imiona i nazwiska:* | **Witold Marciniak 226194,**<br>**Kacper Szymula 226215** |
| *Grupy:* | **Czwartek TP 11:15 (W. Marciniak),**<br>**Czwartek TN 11:15 (K. Szymula)** |
| *Prowadzący:* | **dr inż. Tadeusz Tomczak** |


Wrocław 2019

# Spis treści

# Spis rysunków



# Kody źródłowe

# 1 Wstęp

## 1.1 Cel

Zadaniem projektowym było zaimplementowanie biblioteki umożliwiająca przeprowadzanie operacji arytmetycznych na liczbach dowolnej precyzji. Warunkiem koniecznym była konwersja liczby do wewnętrznej reprezentacji w systemie U2.

## 1.2 Założenia projektowe

- Aplikacja będzie zaimplementowana w języku C++ w wersji 14.
- Zostaną przeprowadzone testy jednostkowe, czasowe oraz profilowanie.
- Za bibliotekę uznajemy zbiór przeciążonych operatorów oraz funkcje wspomagające operacje.
- Operacje arytmetyczne, które zostaną zaimplementowane to operacje dodawania, odejmowania, mnożenia i dzielenia.
- Zostaną również zaimplementowane funkcje wspomagające operacje arytmetyczne takie jak przesunięcia bitowe oraz negacja.
- Wewnętrzną reprezentacją liczby będzie wektor zmiennych typu unsigned int. Zmienna typu unsigned int jest interpretowany przez procesor jako wartość zapisana w pozycyjnym systemie dwójkowym (U2).
- Dowolna precyzja jest ustalona z góry przez użytkownika. Nie należy mylić z nieograniczoną precyzją.
- Pozycja podziału na część całkowitą i część ułamkową (przecinek) w liczbie stałoprzecinkowej jest ustalana w konsekwencji do ustalonej precyzji.
- Operacje arytmetyczne będą wykonywane na dwóch liczbach tej samej precyzji. Wynik tych operacji również będzie liczbą o identycznej precyzji.

## 1.3 Środowisko

Aplikacja została stworzona przy użyciu Visual Studio 2015 (system operacyjny Windows 10) oraz Gedit (system Linux dystrybucja Ubuntu 16.04). Testy zostały przeprowadzone na systemie operacyjnym Linux (dystrybucja Ubuntu 16.04).

## 1.4 Sposób podejścia do problemu

Realizacja projektu odbywała się w następujących etapach (nasza ocena zastosowanego podejścia przedstawiona została w podsumowaniu projektu):

1. Wybór tematu.
2. Sformułowanie założeń projektowych.
3. Próby zaproponowania rozwiązania zgodnie z wiedzą nabytą na wcześniejszych etapach studiów (występowały błędy w rozumieniu oczekiwanych efektów oraz braki w postawach teoretycznych).
4. Konsultacje proponowanych rozwiązań z prowadzącym.
5. Nabywanie nowych wiadomości poprzez sięgniecie do dokumentacji, źródeł dostępnych w internecie oraz konsultacji z prowadzącym.
6. Zmiana koncepcji realizacji projektu w oparciu o nowe wiadomości.
7. Akceptacja proponowanego rozwiązania przez prowadzącego oraz zdefiniowanie ograniczeń zaakceptowanego rozwiązania.
8. Implementacja projektu.
9. Stworzenie sprawozdania projektowego i własnej oceny efektów pracy.

# 2 Wykonanie projektu

Podczas realizacji projektu zostały zaimplementowane zaplanowane operacje. Wygenerowany został też kod bajtowy, przeprowadzone zostały testy jednostkowe, ocena wyprodukowanego kodu bajtowego, pomiary wydajności oraz operacje profilowania kodu.

## 2.1 Decyzje projektowe

W konsekwencji wiedzy nabytej podczas przygotowań do implementacji do projektu oraz konsultacji zostały podjęte następujące decyzje.

1. W celu zapewnienia możliwości wprowadzenia dowolnej wartości, liczba będzie wprowadzana w formacie String. Dopuszczalnymi znakami są cyfry z zakresu 0-9 oraz znaki: '-' i '.'.

2. Zakładamy, że wprowadzane dane są poprawne. Weryfikacja nie została zaimplementowana.

3. Precyzja liczby jest ustalana przez użytkownika. Użytkownik może ustawić dowolną precyzję, jednak musi być ona ustalona przed wprowadzeniem liczby.

4. Wewnątrz biblioteki liczba jest reprezentowana przez wektor zmiennych typu unsigned int, każdy element wektora ma 4 bajty czyli 32 bity.

5. Część ułamkowa jest zapisywana na wcześniej ustalonej liczbie bitów. W przypadku nieskończonego rozwinięcia części ułamkowej po wypełnieniu wszystkich bitów wartość jest zaokrąglana przez obcięcie.

6. Częśc całkowita jest zapisywana analogicznie do części ułamkowej. W przypadku wartości liczby przekraczającej wybraną precyzje cyfry na młodszych pozycjach nie podlegają konwersji, a w konskwencji dostajemy błędną reprezentacje liczby.

## 2.2 Zaimplementowane operacje

W wyniku wykonanej pracy udało się zaimplementować klasę, zbiór funkcji oraz przeciążonych operatorów realizujących następujące operacje:

- konwersja liczby

- dodawanie
- odejmowanie
- mnożenie
- dzielenie
- przesunięcie bitowe
- ustawianie wartości liczby (wykorzystane do testów jednostkowych)

## 2.3 Opisy algorytmów

W tej części sprawozdania zostały zawarte opisy kluczowych algorytmów dla biblioteki. Algorytmy operacji pomocniczych nie zostały zawarte.

### 2.3.1 Konwersja ze String do U2

Algorytm konwersji ze String do U2 został zaimplementowany w oparciu o wariacje algorytmu Hornera. Pseudo kod pokazany poniżej:

Kody źródłowe 1: Pseudokod algorytmu konwersji

```
Set next_additive to 0.
For every digit in number (starting at the left):
    Set additive to next_additive.
    If the digit is odd, set next_additive to 5, else set it to 0.
    Divide the digit by two (truncating) then add additive.
Remove leading zero if necessary (if it starts with 0 but is not just 0).
```

Poniżej właściwy kod użyty w programie:

Kody źródłowe 2: Kod algorytmu konwersji

```
//CZESC CALKOWITA
        //dopoki jest reszta i wynik dzielenia to liczymy
        //bo to znaczy ze sa jeszcze bity do zakodowania
        for(int b = mf.dnb; div_total_info != 0 || rest != 0; b++)
        {
                div_total_info = 0; //zerujemy po wejsciu
                additive = 0;
                for(int i = 0; i < c_part.size(); i++)
```

```
{
        digit = (int)c_part[i] - 0x30;
        rest = digit & 1;  //reszta z dzielenia przez 2
        div_res = digit>>1; //dzielenie przez 2;
        new_digit = div_res + additive;
        c_part[i] = new_digit + 0x30;
        //gdy jest jakas cyfra co jest inna niz zero
        if(new_digit != 0) div_total_info = 1;
        //obliczamy nowy dodatek
        if(rest) additive = 5;
        else  additive = 0;
}
//wiedzac ze b jest aktualnie uzupelnianym bitem w liczbie to
//musimy obliczyc numer segmentu dla tego bitu
int seg = b/(sizeof(seg_t)*8);
int pos = b%(sizeof(seg_t)*8);
//kodujemy bit w liczbie
mf.bits[seg] |= ( ((seg_t)rest)<<pos );
```

W sprawozdaniu zawarta została tylko konwersja części całkowitej. Konwersja części ułamkowej, została oparta o te samą wariacje schematu Hornera. Algorytm wyprowadzania stworzony został analogicznie z tą różnica ze operacje miału odwróconą kolejność.

### 2.3.2 Dodawanie i odejmowanie

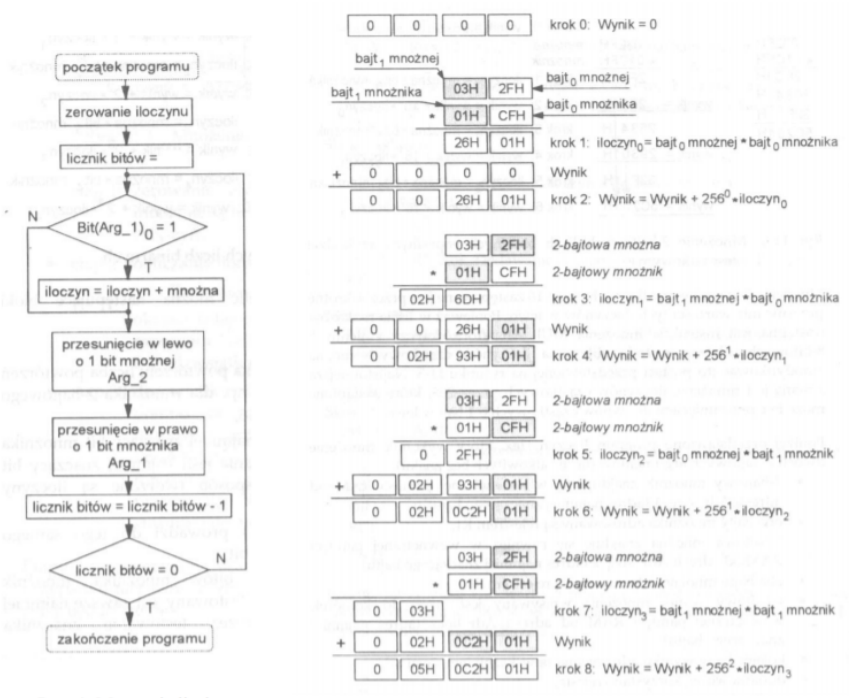
Dodawanie i odejmowanie liczb wielobajtowych polega na dodawaniu kolejnych bajtów wektora zmiennych (zaczynając od najmniej znaczącego bajtu) z uwzględnieniem przepełnienia (pożyczkę), które mogło nastąpić po dodawaniu (odejmowaniu) poprzednich bajtów. W naszym przypadku sprawdzanie przepełnia odbywało się przy użyciu instrukcji warunkowej i dodatkowej zmiennej o wartości równej wartości przepełnienia (pożyczki). Konieczne było rzutowaniu elementów wektora na zmienne o większej "pojemności"w celu zweryfikowania wystąpienia przepełnienia (pożyczki).
Kod dodawania poniżej, odejmowanie zostało zrealizowane analogicznie:

Kody źródłowe 3: Kod algorytmu dodawania

```
myfixedpoint myfixedpoint::operator+(myfixedpoint& mf)
{
	myfixedpoint sum(res_bits, dnb); //taka sama liczba wielkosciowo jak
	    ↪ argumenty
	long long unsigned int sum_seg;
	long long unsigned int a,b,c;
	seg_t carry = 0;

	//sumujemy wszystkie segmenty obydwu liczb, biorac pod uwage
	    ↪ przeniesienia
	for(int i = 0; i < bits.size(); i++)
	{
		a = bits[i] & MAX_SEGMENT;
		b = mf.bits[i] & MAX_SEGMENT;
		c = carry & MAX_SEGMENT;

		sum_seg = (a + b + c);

		//sprawdzamy czy bylo przenieisenie
		if(sum_seg & CARRY_SUM_BIT) carry = 1;
		else		carry = 0;

		//wynik
		sum.bits[i] = (seg_t)sum_seg;
	}
	return sum;
}
```

### 2.3.3 Mnożenie

Algorytm mnożenia został zaimplementowany w oparciu o schemat:




Rysunek 1: Schemat blokowy mnożenia

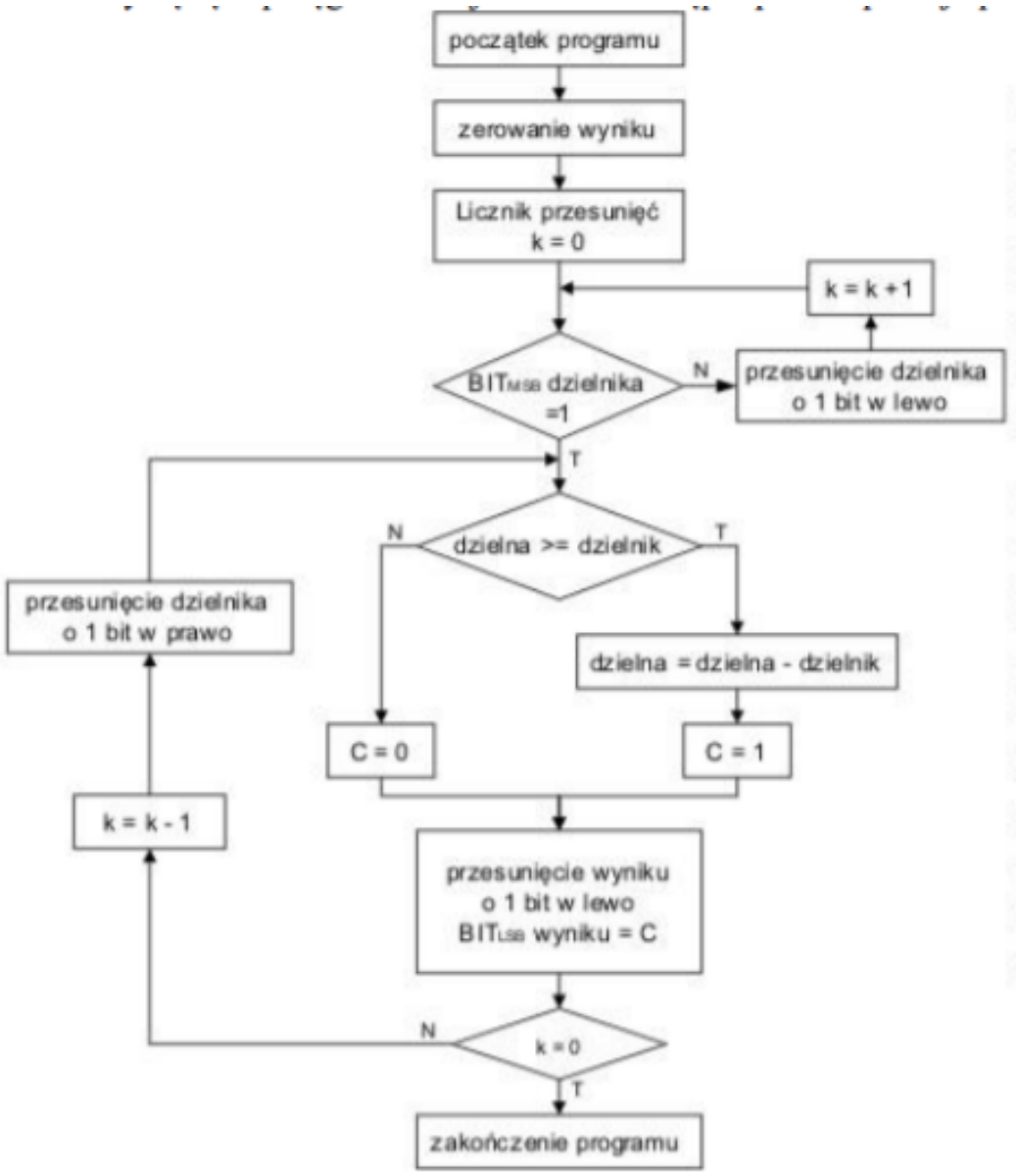
Schemat przedstawia realizacje operacji mnożenia poprzez dodawanie do siebie odpowiednio przesuniętej (pomnożonej przez potęgę liczby 2) mnożnej. W naszym algorytmie wykorzystane zostały operacje pomocnicze definiujące znak liczby oraz funkcje wykonujące odpowiednie przesunięcie. W sprawozdaniu zostały pominięte fragmenty definiujące znak liczby oraz niektóre komentarze.

Kody źródłowe 4: Kod algorytmu mnożenia

```
myfixedpoint myfixedpoint::operator*(myfixedpoint& mf)
{
        myfixedpoint multiplicand(res_bits, dnb);  //mnozna
        myfixedpoint multiplier(res_bits, dnb);   //mnoznik
        myfixedpoint product(res_bits, dnb);   //iloczyn
        int bit;
        int minus = 0;
        multiplicand = *this;
        multiplier = mf;
        for(int b = 0; b < multiplier.res_bits; b++)
        {
                myfixedpoint sum = multiplicand;   //sumator pomocniczy
                int seg = b/(sizeof(seg_t)*8);
                int pos = b%(sizeof(seg_t)*8);
                //pobieramy wartoscbitu
                if(multiplier.bits[seg] & (1<<pos))  bit = 1;
                else         bit = 0;
                if(bit)
                {
                        sum.shift_left(b);
                        product = product + sum;  //do glownego wyniku
                }
        }
        product.shift_right(dnb);
        return product;
}
```

### 2.3.4 Dzielenie

Algorytm dzielenia został zaimplementowany w oparciu o schemat:




Rysunek 2: Schemat blokowy dzielenia

Schemat przedstawia realizacje operacji dzielenia poprzez odejmowanie do dzielnej odpowiednio przesuniętego dzielnika. Sprawdzamy ile razy dzielnik zmieści się w dzielnej i uzupełniamy odpowiednie bity wyniku. W naszym algorytmie wykorzystane zostały operacje pomocnicze definiujące znak liczby

oraz funkcje wykonujące odpowiednie przesunięcie. Algorytm nie jest algorytmem dokładnym, przybliżenie wyniku będzie zależało od ustalonej precyzji. W sprawozdaniu zostały pominięte fragmenty definiujące znak liczby oraz niektóre komentarze.

Kody źródłowe 5: Kod algorytmu dzielenia

```
myfixedpoint myfixedpoint::operator/(myfixedpoint& mf)
{
	myfixedpoint dividend(res_bits, dnb);   //dzielna
	myfixedpoint divider(res_bits, dnb);   //dzielnik
	myfixedpoint quotient(res_bits, dnb);   //iloraz
	myfixedpoint sub_temp(res_bits, dnb);   //zmienna pomocniza
	int bit;
	int k = 0;
	int minus = 0;
	int end = 0;
	dividend = *this;
	divider = mf;
	dividend.shift_left(dnb);
	while(end == 0)
	{
		if(divider.bits[divider.bits.size()-2] & (1<<(sizeof(seg_t)*8-2))
		    ↪ )
		{
			end = 1;
		}
		else
		{
			divider.shift_left(1);
			k++;  //zwiekszamy licznik przesuniec
		}
	}
	for(; k >= 0; k--)
	{
		sub_temp = dividend - divider;
		if(sub_temp.bits[sub_temp.bits.size()-1] & (1<<(sizeof(seg_t)
		    ↪ *8-1)))
		{
			bit = 0;
```

```
                }
                else
                {
                        bit = 1;
                        dividend = dividend - divider;
                }
                quotient.shift_left(1);  //przesuwamy w lewo o 1
                quotient.bits[0] |= bit; //na najmlodsza pozycje ustawiamy
                    ↪ nowy zakodowany bit
                divider.shift_right(1);
        }
        return quotient;
}
```

# 3 Przeprowadzone Testy

## 3.1 Testy jednostkowe

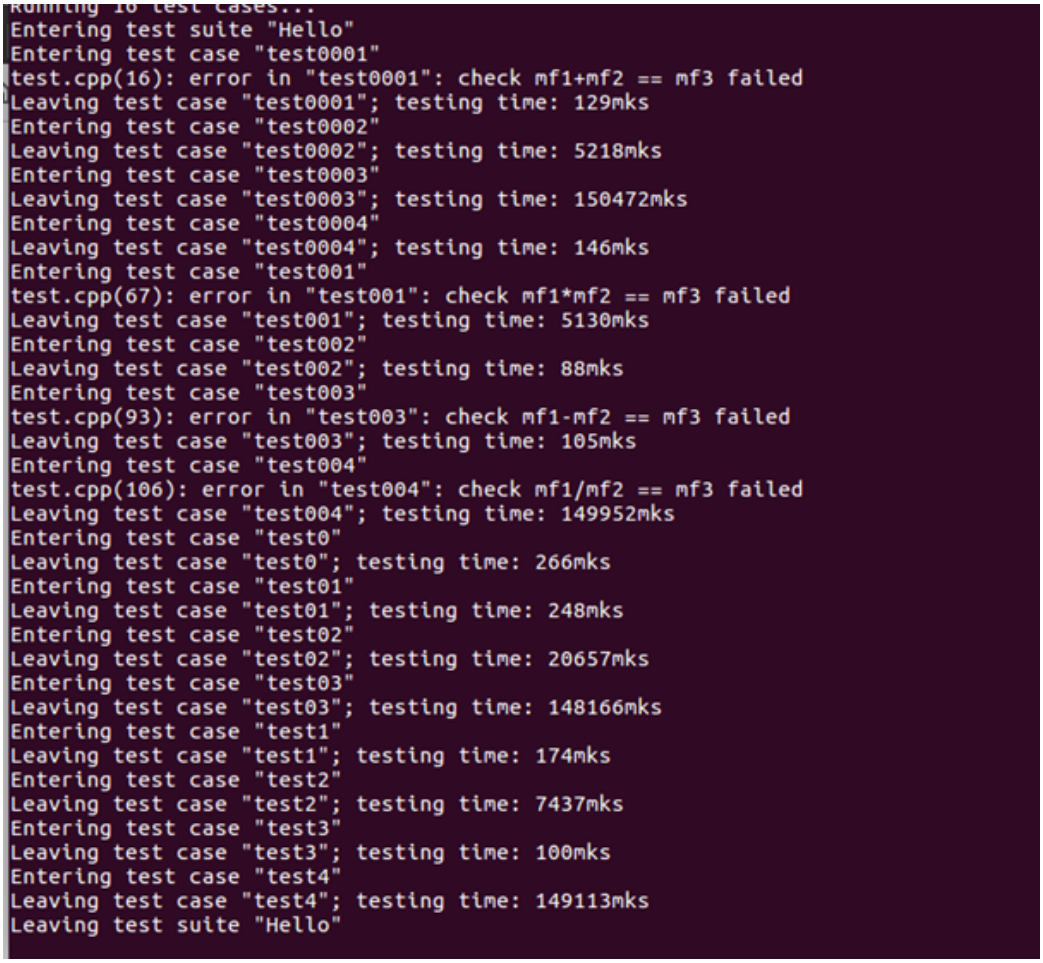
Do tworzenia testów jednostkowych w systemie operacyjnym linux użyliśmy biblioteki Boost. Jest prosta w obsłudze, łatwo konfigurowalna oraz umożliwia wiele funkcji. Weryfikowaliśmy naszą bibliotekę pod względem poprawnego działania operacji zaimplementowanych podczas projektu. Testowaliśmy instrukcje dodawania, odejmowania, mnożenia oraz dzielenia. Przykładowe testy operacji poniżej:

Kody źródłowe 6: Testy jednostkowe

```
BOOST_AUTO_TEST_CASE( test0)
{
        myfixedpoint mf1(10264,64);
        myfixedpoint mf2(10264,64);
        mf1.setValue("123456789123456789123456789333.913123123");
        mf2.setValue("18293101258125912471249124912491.31239129912");
        myfixedpoint mf3(10264,64);
        mf3.setValue("125286099249269380370581701825.
        225514422119999999923750655383969 3
        879126571118831634521484375");

        BOOST_CHECK( mf1+mf2 == mf3);

}
BOOST_AUTO_TEST_CASE( test01)
{
        myfixedpoint mf1(10264,64);
        myfixedpoint mf2(10264,64);
        mf1.setValue("123456789123456789123456789333.913123123");
        mf2.setValue("18293101258125912471249124912491.31239129912");
        myfixedpoint mf3(10264,64);
        mf3.setValue("121627478997644197876331876842.
        600731823880000000001861133
        913649837268167175352573394775390625");

        BOOST_CHECK( mf1-mf2 == mf3);

}
```

```
BOOST_AUTO_TEST_CASE( test02)
{
        myfixedpoint mf1(10264,64);
        myfixedpoint mf2(10264,64);
        mf1.setValue("12345678912345678912345678933 3.913123123");
        mf2.setValue("18293101258125912471249124 91.31239129912");
        myfixedpoint mf3(10264,64);
        mf3.setValue("22584075444384928558339646405180 45
        92194469530983296659244.48978716675124519099385200782137 4
        17263584211468696594238281 25");

        BOOST_CHECK( mf1*mf2 == mf3);

}
BOOST_AUTO_TEST_CASE( test03)
{
        myfixedpoint mf1(10264,64);
        myfixedpoint mf2(10264,64);
        mf1.setValue("12345678912345678912345678933 3.913123123");
        mf2.setValue("18293101258125912471249124 91.31239129912");
        myfixedpoint mf3(10264,64);
        mf3.setValue("67.488168015588113857639
        46358307130424236675025895237922668457031 25");

        BOOST_CHECK( mf1/mf2 == mf3);

}
```

Napisanie testu wiążę się z utworzeniem obiektu o konkretnej precyzji. Następnie „ustawieniu" wartości liczbowej w każdym obiekcie, wykonanie danej operacji arytmetycznej i porównanie wyników czy się ze sobą zgadzają przy pomocy operator przypisania zaimplementowanego do przeprowadzania testów jednostkowych.
Na poniższym zdjęciu widzimy wycinek konsoli z wynikami testów.




Rysunek 3: Testy jednostkowe

## 3.2 Profilowanie

Dokonaliśmy profilowania kodu przy pomocy narzędzia gprof. Aby uzyskać plik umożliwiający wydrukowanie naszego pliku z wynikami profilowania należało nasz program skompilować z flagą -pg oraz włączyć stworzony plik wykonywalny, po czym wygenerował się plik tekstowy z konkretnymi wartościami zawartymi poniżej. Umieściliśmy ważniejsze części tegoż pliku, gdyż są one najbardziej znaczące w kwestii całego programu. Narzędzie gprof pierw sortuje wyniki po czasie spędzonym w programie wyrażonym w procentach, następnie po ilości wywołań, a na końcu sortuje wyniki alfabetycznie.

Kody źródłowe 7: Profilowanie

```
Flat profile:
Each sample counts as 0.01 seconds.
  % cumulative self self total
 time seconds seconds calls ms/call ms/call name
 40.02 0.04 0.04 42552700 0.00 0.00 std::vector<unsigned int, std::allocator<
     ↪ unsigned int> >::operator[](unsigned long)
 20.01 0.06 0.02 10141 0.00 0.00 myfixedpoint::operator-(myfixedpoint&)
 10.00 0.07 0.01 20281 0.00 0.00 myfixedpoint::shift_left(int)
 10.00 0.08 0.01 10154 0.00 0.00 __gnu_cxx::__enable_if<std::__is_scalar<
     ↪ unsigned int>::__value, unsigned int*>::__type std::__fill_n_a<
     ↪ unsigned int*, unsigned long, unsigned int>(unsigned int*, unsigned
     ↪ long, unsigned int const&)
 10.00 0.09 0.01 10140 0.00 0.00 myfixedpoint::shift_right(int)
  5.00 0.10 0.01 6627727 0.00 0.00 std::vector<unsigned int, std::allocator<
     ↪ unsigned int> >::size() const
  5.00 0.10 0.01 10143 0.00 0.00 std::remove_reference<std::vector<unsigned int
     ↪ , std::allocator<unsigned int> >&>::type&& std::move<std::vector<
     ↪ unsigned int, std::allocator<unsigned int> >&>(std::vector<unsigned
     ↪ int, std::allocator<unsigned int> >&)
```

## 3.3 Wyprodukowanie kodu bajtowego

Wyprodukowany został również kod bajtowy dla poszczególnych funkcji, który przeanalizowaliśmy w celu sprawdzenia użytych mechanizmów procesora oraz szukania możliwych sposobów poprawy wydajności. Ze względu na swoją objętość nie został on zawarty w sprawozdaniu.

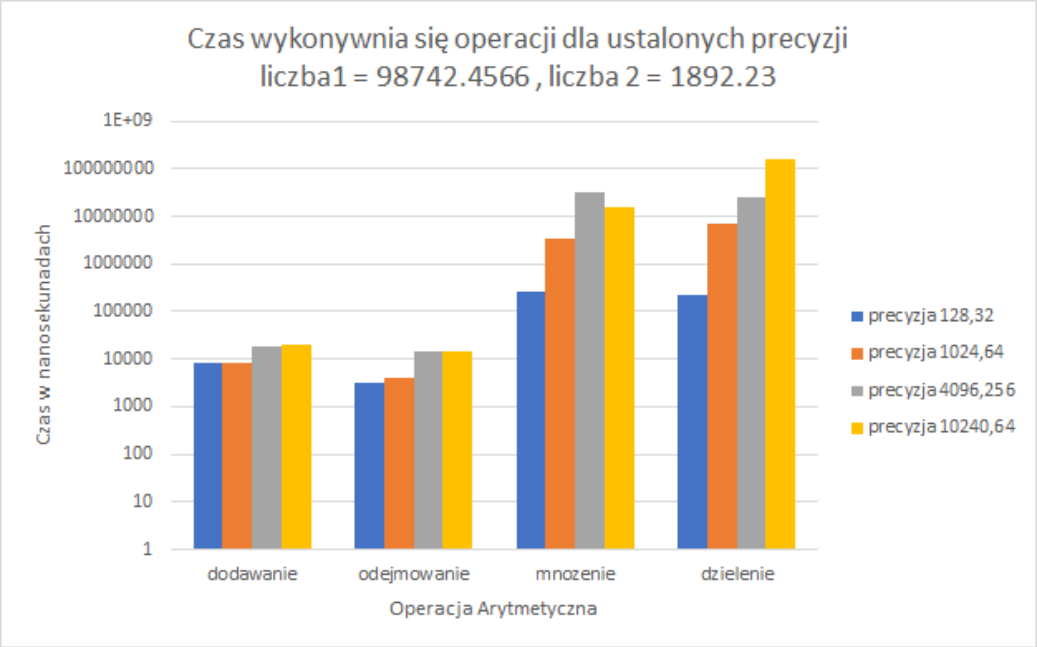
## 3.4 Testy czasowe

Pomiary czasowe zostały przeprowadzone przy użyciu biblioteki chrono.

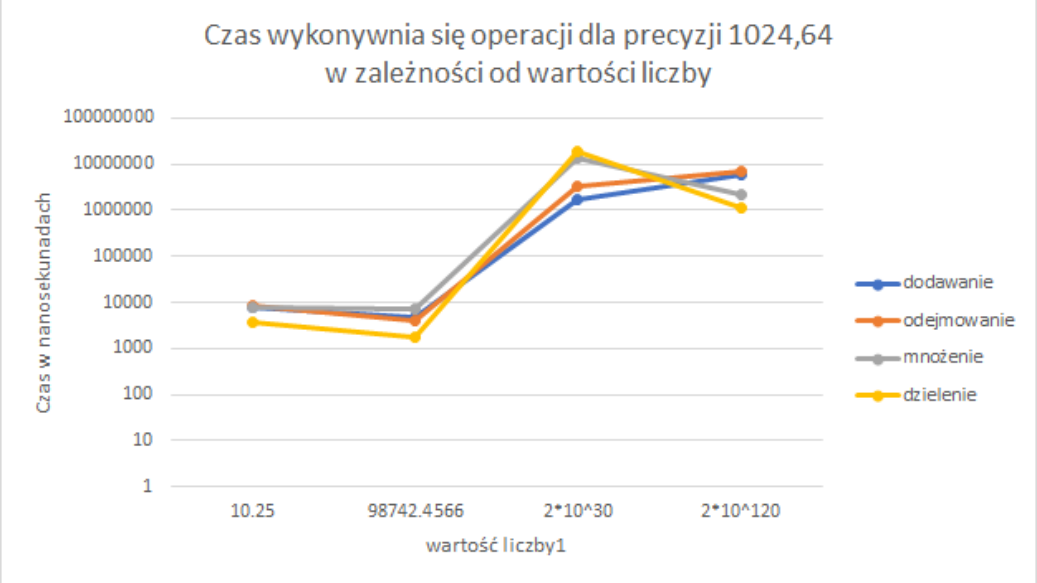
| | | | Czas w nanosekundach. | | | | |
|---|---|---|---|---|---|---|---|
| Nr pomiaru | liczba1 | liczba2 | dodawanie | odejmowanie | mnozenie | dzielenie | precyzja |
| 1 | 10.25 | 2.80 | 7057 | 3177 | 210257 | 204644 | 128,32 |
| 2 | 98742.4566 | 1892.23 | 8455 | 3128 | 260272 | 218070 | 128,32 |
| 3 | 2*10^30 | 3*10^45 | 2833 | 1385 | 239138 | 9873 | 128,32 |
| 4 | 2*10^120 | 1.23*10^100 | 2176 | 880 | 230235 | 4108 | 128,32 |
| 5 | 10.25 | 2.80 | 7875 | 4617 | 1714806 | 5699251 | 1024,64 |
| 6 | 98742.4566 | 1892.23 | 8376 | 3952 | 3416036 | 7000386 | 1024,64 |
| 7 | 2*10^30 | 3*10^45 | 8054 | 7071 | 13049179 | 2277907 | 1024,64 |
| 8 | 2*10^120 | 1.23*10^100 | 3740 | 1743 | 18426766 | 1091299 | 1024,64 |
| 9 | 10.25 | 2.80 | 9249 | 9644 | 29041806 | 24730719 | 4096,256 |
| 10 | 98742.4566 | 1892.23 | 17805 | 14585 | 31766738 | 25000803 | 4096,256 |
| 11 | 2*10^30 | 3*10^45 | 18803 | 15131 | 30527978 | 24483469 | 4096,256 |
| 12 | 2*10^120 | 1.23*10^100 | 10005 | 7655 | 77901021 | 23481998 | 4096,256 |
| 13 | 10.25 | 2.80 | 39010 | 28329 | 12041471 | 157480914 | 10240,64 |
| 14 | 98742.4566 | 1892.23 | 20833 | 14890 | 15979400 | 158861794 | 10240,64 |
| 15 | 2*10^30 | 3*10^45 | 21791 | 14822 | 31896330 | 162691569 | 10240,64 |
| 16 | 2*10^120 | 1.23*10^100 | 21481 | 14670 | 104480667 | 163138484 | 10240,64 |

Rysunek 4: Pomiary czasowe

W teście numer 3 i 4, dla operacji mnożenia i dzielenia została przekroczona precyzja dlatego operacje dały błędny wynik, a testy czasowy są nie miarodajny dlatego zostaną pominięte przy formułowaniu wniosków. Poniżej przedstawione zostały wykresy pokazujące wydajność biblioteki w zależności od ustalonej precyzji oraz wartości liczby:




Rysunek 5: Wykres pomiaru czasu w zależności od precyzji




Rysunek 6: Wykres pomiaru czasu w zależności wartości liczb

# 4 Analiza wyników

Analiza przeprowadzonych testów i pomiarów doprowadziła nas do następujących wniosków:

- Można śmiało stwierdzić, że nie wszystkie testy jednostkowe przeszły pomyślnie. Chcieliśmy sprawdzić nie tylko poprawność wykonywanych działań, a również sytuacje w których przetestujemy wartości sprzeczne z poprawnymi wynikami operacji arytmetycznych. Testy te pokazały ograniczenia w funkcjonowaniu biblioteki. Dla przykładu dla liczb z częścią ułamkową o nieskończonym rozwinięciu wynik będzie niedokładny. Niedokładność w takim przypadku będzie rzędu 1/precyzja części ułamkowej.

- Jak widać po wyniku profilowania, nasz program najwięcej czasu „spędza" jak i najczęściej odwołuje się do vectora z biblioteki STL. Dużo czasu też nasz program spędza w funkcjach służących do przesunięć bitowych. Jest ono używane w większości operacji arytmetycznych więc będzie często używane.

- Pomiary czasu pokazały, że wraz ze wzrostem ustalonej precyzji rośnie czas wykonywania działań. Dla mnożenia i dzielenia czas rośnie odpowiednio szybciej ponieważ operacje te wykorzystują operacje dodawania i odejmowania.

- Czas wykonywania się poszczególnych operacji nie jest do końca zależny od wartości dziesiętnej liczby. Wartość dziesiętna ma wpływ na liczbę przesunięć bitowych, koniecznych do wykonania obliczeń, ale to liczba zer i jedynek w reprezentacji U2 liczby jest bardziej odpowiedzialną za definiowanie czasu potrzebnego do wykonania operacji, ponieważ to od niej zależy liczba wykonanych instrukcji.

- Wyprodukowany kod bajtowy był tak rozległy, że trudno było go porównać ze znalezionymi w internecie rozwiązaniami. Dało się jednak zauważyć, że w porównaniu do kodów realizujących podobne operacje produkujemy zostało wyprodukowane bardzo dużo nadmiarowego kodu związanego z użyciem wektor co pozwala poddawać w wątpliwość jego zastosowanie.

# 5 Uzasadnienie zastosowanych rozwiązań

1. Precyzja. Podjęliśmy decyzje o precyzji liczb ustalanej przez użytkownika. Do zalet takiego rozwiązania należą:

   - Przyśpieszenie operacji - jeżeli znalibyśmy przybliżony zakres wartości liczb moglibyśmy szybciej wykonywać operacje.
   - Eliminacja problemu nieskończonych rozwinięć części ułamkowej.
   - Brak konieczności "przesuwania przecinka"dla liczb o różnej precyzji.
   - Brak konieczności wielokrotnego alokowania pamięci.

   Wadami takiego rozwiązania są:

   - Alokowanie zbyt dużych obszarów pamięci w stosunku do potrzebnych do wykonania operacji.
   - Błędy obliczeniowe w przypadku przekroczenia precyzji.
   - Niedokładności w przybliżeniach części ułamkowej co wpływa na wypaczenie wyniku.

2. Reprezentacja wewnętrzna. Podjęliśmy decyzje o reprezentacji wewnętrznej w formie wektora zmiennych typu unsigned int. Do zalet takiego rozwiązania należą:

   - Możliwość użycia funkcji z biblioteki vector.
   - Możliwość podejrzenia "bitowej reprezentacji: przechowywanej przez wektor.
   - Możliwość rzutowania zmiennej na zmienną o większym rozmiarze co jest niezbędne przy operacjach arytmetycznych.

   Wadami takiego rozwiązania są:

   - Duża ilość pobocznych operacji procesora wyprodukowanych przez kompilator.
   - Duży czas wykonywania się operacji.

3. Algorytmy operacji arytmetycznych.

- Dodawanie. Klasyczny algorytm dodawania pozycyjnego. Wadą naszego rozwiązania jest konieczność użycia dodatkowych zmiennych do weryfikacji konieczności przeniesienia.
- Odejmowanie. Klasyczny algorytm odejmowania pozycyjnego. Wadą naszego rozwiązania jest konieczność użycia dodatkowych zmiennych do weryfikacji konieczności pożyczki.
- Mnożenie. Algorytm nie jest optymalny. Wadą jest duża ilość koniecznych do wykonania operacji oraz alokowanie pamięci na zmienne pomocnicze.
- Dzielenie. Algorytm nie jest dokładny oraz optymalny. Wadą jest duża ilość koniecznych do wykonania operacji oraz alokowanie pamięci na zmienne pomocnicze.

4. System wprowadzania liczb. Podjęliśmy decyzje o wprowadzaniu liczb w systemie dziesiętnym. Do zalet takiego rozwiązania należą:

   - Intuicyjność. Przy wykonywaniu obliczeń automatycznie sprawdzamy poprawność wyniku w systemie dziesiętnym.
   - Duża ilość podobnych rozwiązań dająca możliwość porównania.

   Wadami takiego rozwiązania są:

   - Trudność w konwersji do systemu U2.
   - Występowanie liczb o nieskończonym rozwinięciu.

# 6 Proponowane metody ulepszenia zaproponowanych rozwiązań

1. Co łatwo wywnioskować z profilowania, w przypadku zoptymalizowania kodu można by było ulepszyć kod tworząc własny szablon klasy, a nie używać gotowych rozwiązań dostarczonych przez bibliotekę vector.

2. W celu ułatwienia funkcjonowania biblioteki można by wprowadzać liczby w systemie szesnastkowym. Ułatwiłoby to znacznie konwersje do systemu U2. Moglibyśmy wykorzystać bazy skojarzone.

3. Operacje dodawania i odejmowania można by usprawnić wykorzystując mechanizmy procesora propagujące przeniesienia lub pożyczki.

4. Algorytmy mnożenia można zoptymalizować stosując algorytmy Bootha, Bootha-McSorleya zamiast wykorzystywać algorytm mnożenia pozycyjnego.

5. Dzielenie powinno być zrealizowane algorytmem pozwalającym dzielić, aż do uzyskania liczby żądanej precyzji.

6. Biblioteka powinna sama ustalać precyzje odpowiednia dla wprowadzonej liczby.

7. Można by wykorzystać operacje przesunięć bitowych wspierane przez mechanizmy procesora. Wiązałoby się to z przedefiniowaniem struktury wewnętrznej reprezentacji.

8. Dobrym pomysłem byłoby utworzenie mechanizmu walidacji wprowadzanych danych.

9. Bibliotekę można by rozbudować tak aby umożliwiała operacje na liczbach o różnych precyzjach.

# 7 Podsumowanie i Wnioski

Podsumowując realizacja projektu zwiększyła naszą wiedzę dotyczącą architektury procesora oraz przedefiniowała myślenie dotyczące sposobu reprezentacji liczb. Wykonana biblioteka posiada niestety zbyt wiele ograniczeń i niedoskonałości do użytku komercyjnego, jednak wytworzenie jej było dla nas ciekawym wyzwaniem programistycznym. Z perspektywy wykonanej pracy stwierdzamy, że nasz sposób podejścia do problemu był błędny i gdyby nie jego weryfikacja to realizacja zakończyłaby się całkowitym niepowodzeniem. Na etapie początkowym brakowało nam informacji i wiedzy aby próbować rozwiązywać problem metodą prób i błędów. Najwięcej problemu dostarczyło nam zrealizowanie operacji konwersji do wewnętrznej reprezentacji U2, ujawniła się tu kwestia złego zrozumienia oczekiwań projektowych oraz brak wiedzy o tym jak procesor przetwarza liczby.

## 7.1 Dalsze możliwości rozwoju projektu

Następnym krokiem rozbudowy biblioteki mogłaby być poprawa istniejących funkcjonalności, uporanie się z ograniczeniami narzuconymi przez wybraną precyzję oraz zaimplementowanie nowych operacji takich jak potęgowanie i pierwiastkowanie.

# 8 Bibliografia


1. IA-32 Intel® Architecture Software Developer's Manual
2. POLITECHNIKA OPOLSKA Instytut Automatyki i Informatyki Laboratorium architektury komputerów Algorytmy mnożenia i dzielenia wielobajtowego
3. https://stackoverflow.com/questions/11006844/convert-a-very-large-number-from-decimal-string-to-binary-representation
4. https://stackoverflow.com/questions/50554877/how-to-use-processor-instructions-in-c-to-implement-fast-arithmetic-operations


## 8.1 Link do repozytorium

`https://github.com/Kapeo/OiAK_project`