Organizacja i architektura komputerów

Implementacja procedur obliczeń na liczbach zmiennoprzecinkowych za pomocą instrukcji stałoprzecinkowych

Projekt wykonali:

Ireneusz Żwirek 235073
Adam Maciak 226113

Grupa:
Czw. TP 13:15

Prowadzący:

Dr inż. Tadeusz Tomczak

Spis treści

Spis fragmentów kodu

Spis rysunków

1. Przedstawienie zadania

Zadaniem projektowym była implementacja procedur obliczeń zmiennoprzecinkowych na liczbach o rozmiarze 80 bit za pomocą instrukcji stałoprzecinkowych korzystającym z dowolnego słowa w wybranym kompilowanym języku programowania.

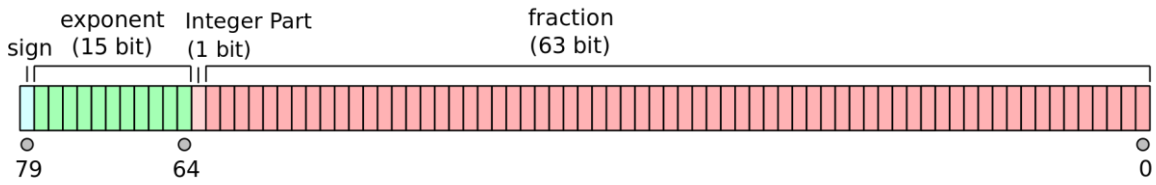
Zadanie zostało zrealizowane w języku c++ pod systemem Linux. Wykorzystane zostały następujące narzędzia programistyczne:

- Visual Studio Code,
- CMake,
- gcc,
- gdb,
- Google Test.

2. Opis implementacji

2.1 Struktura danych

Do przechowywania liczby zmiennoprzecinkowej został wykorzystany format rozszerzonej precyzji X86, którego strukturę ilustruje poniższy rysunek.



Rysunek 1 - Struktura formatu zmiennoprzecinkowego

Na liczbę zmiennoprzecinkową składa się bit znaku, 15 bitów wykładnika oraz 64-bitowa mantysa, której pierwszy bit to część całkowita, zaś pozostałe 63 bity to część ułamkowa.

Struktura została zaimplementowana w postaci własnej klasy ExtendedPrecission, która składa się z następujących pól:

Fragment kodu 1 - pola klasy ExtendedPrecission

```
uint8_t sign;
uint16_t exponent;
uint64_t mantissa;
```

Powyższe pola przechowują kolejno informacje o znaku, wykładniku oraz części znaczącej liczby zmiennoprzecinkowej jako nieoznakowane liczby całkowite rozmiaru dopasowanego do pola liczby.

2.1.1 Konwersja liczby zmiennoprzecinkowej do notacji binarnej

Przy konwersji liczby zmiennoprzecinkowej do binarnego formatu rozszerzonej precyzji został wykorzystany algorytm Malcolma Slaneya i Kena Turkowskiego z 1991 roku. Otrzymany na jego wyjściu ciąg bitów został dopasowany do formatu danych klasy ExtendedPrecission.

2.1.2 Wyświetlenie liczby w postaci binarnej

Liczba typu ExtendedPrecission jest wypisywana w postaci ciągu składającego się z 82 znaków, na który składają się pola klasy w postaci binarnej oddzielone od siebie spacjami.

Fragment kodu 2 - zapis liczby w postaci binarnej

```
std::string ExtendedPrecission::toString()
{
    std::string s = "";
    if (this->sign == 0)
        s += '0';
    else
        s += '1';
    s += " ";
    s += intToBinaryString(this->exponent);
    s += " ";
    s += intToBinaryString(this->mantissa);
    return s;
}
```

2.2 Porównywanie liczb

2.2.1 Operator >

Operator większości zwraca informację, czy liczba poprzedzająca operator jest większa od liczby za operatorem. W pierwszej kolejności porównywane są znaki – jeżeli są różne, to liczba dodatnia jest większa i operator zwraca odpowiednią wartość typu bool. W wypadku takich samych znaków porównywane są wartości wykładników – dla liczb dodatnich większy wykładnik oznacza większą liczbę, w wypadku liczb ujemnych jest przeciwnie. Jeśli to porównanie nie przynosi rozstrzygnięcia, to sprawdzane są części znaczące liczb – tutaj, analogicznie jak przy wykładnikach, większa wartość w liczbach dodatnich oznacza większą liczbę i przeciwnie dla liczb ujemnych.

Fragment kodu 3 - implementacja operatora >

```
bool ExtendedPrecission::operator>(ExtendedPrecission num)
{
    if (this->sign < num.getSign())
        return true;
    else if (this->sign > num.getSign())
        return false;
    else if(this->sign == 0)    //if both >0
    {
        if (this->exponent > num.getExponent())
            return true;
        if (this->exponent < num.getExponent())
            return false;
        return this->mantissa > num.getMantissa();
    }
    else if(this->sign != 0)    // if both <0
    {
        if (this->exponent > num.getExponent())
            return false;
        if (this->exponent < num.getExponent())
            return true;
        return this->mantissa < num.getMantissa();
    }
}
```

2.2.2 Operator <

Operator mniejszości zwraca informację, czy liczba przed operatorem jest mniejsza od liczby następującej po nim. Jego funkcjonalność została zaimplementowana jako dopełnienie funkcjonalności operatorów większości oraz równości – jeżeli pierwsza liczba nie jest ani większa od drugiej, ani jej równa, to zostaje uznana za mniejszą od niej.

Fragment kodu 4 - implementacja operatora <

```
bool ExtendedPrecission::operator<(ExtendedPrecission num)
{
    if (*this > num || *this == num)
        return false;
    return true;
}
```

2.3.3 Operator ==

Operator równości liczb zwraca informację, że liczby są sobie równe, jeżeli wszystkie składowe liczby są równe składowym drugiej liczby.

Fragment kodu 5 - implementacja operatora ==

```
bool ExtendedPrecission::operator==(ExtendedPrecission num)
{
    if (this->sign == num.getSign() &&
        this->exponent == num.getExponent() &&
        this->mantissa == num.getMantissa())
        return true;
    else return false;
}
```

2.3 Dodawanie i odejmowanie

Dodawanie i odejmowanie liczb zostały zaimplementowane w oparciu o wzór

$$W = (\pm M_1 * \beta^{W_1}) \pm (\pm M_2 * \beta^{W_2})$$

Do obliczeń stałoprzecinkowych część znacząca liczby o mniejszym wykładniku jest skalowana w górę(znacząca jest przechowywana w zmiennej typu uint64, zaś bufor obliczeń w int128). Po wykonanym dodawaniu/odejmowaniu znacząca liczba jest skalowana z powrotem do standardu 80 bit.

W programie zostały zaimplementowane obliczenia dla dodawania i odejmowania tak, że zależnie od znaków liczb poddawanym działaniom liczby są przekierowywane do odpowiednich funkcji obliczania znaczących oraz wykładników, a znak jest przydzielany zależnie od znaków liczb lub wyniku działań.

Poniższy fragment przedstawia urywek kodu dla przypadku dodawania dwóch liczb o takim samym znaku – jeżeli są one dodatnie, to wynik będzie dodatni, a wykładniki oraz znaczące zostaną obliczone w dalszej części algorytmu, w funkcji scaleMantissa. Analogiczna sytuacja występuje w wypadku dodawania liczb ujemnych.

Fragment kodu 6 - fragment implementacji dodawania

```
if (this->sign == num.getSign())
    {
        result.sign = this->sign;
        if(this->exponent > num.getExponent())
            result.exponent = this->exponent;
        else
            result.exponent = num.getExponent();

        result.mantissa = scaleMantissa(*this, num, OperationType::osum);
    }
```

Funkcja scaleMantissa oblicza i zwraca wartość części znaczącej w operacjach dodawania i odejmowania(obliczenia są wybierane na podstawie zmiennej OperationType i są analogiczne w implementacji). Jest implementacja powyższego wzoru z wyciągnięciem części wspólnej wykładników przed nawias, zmienna scaledMantissa staje się tymczasowym wynikiem o długości 128 bitów (aby uniknąć przepełnienia. Następuje sprawdzenie, czy nie wystąpiło przeniesienie. w kolejnym kroku zmienna zostaje przeskalowana i rzutowana do wyniku mantysy typu uint64. Funkcja wygląda analogicznie dla odejmowania (przekierowanie warunkiem op==OperationType::osub).

Fragment kodu 7- fragment funkcji scaleMantissa

```
uint64_t scaleMantissa(ExtendedPrecission num1, ExtendedPrecission num2, OperationType op)
{
    __int128_t scaledMantissa;
    int64_t expDiff;

    if (num1.getExponent() > num2.getExponent())
        expDiff = num1.getExponent() - num2.getExponent();
    else
        expDiff = num2.getExponent() - num1.getExponent();

    if (op == OperationType::osum)
    {
        if (num1.getExponent() > num2.getExponent())
        {
            scaledMantissa = pow(2, num2.getExponent() - exponentWeight) * (num1.getMantis
* pow(2, expDiff) + num2.getMantissa());
            if(scaledMantissa > pow(2, num1.getExponent() - exponentWeight + 63))
                carry++;
        }
        else
        {
            scaledMantissa = pow(2, num1.getExponent() - exponentWeight) * (num2.getMantis
* pow(2, expDiff) + num1.getMantissa());
            if(scaledMantissa > pow(2, num2.getExponent() - exponentWeight + 63))
                carry++;
        }

        while (scaledMantissa > UINT64_MAX)
        {
            scaledMantissa = scaledMantissa/2;
        }
    }
    scaledMantissa = static_cast<uint64_t>(scaledMantissa);
    return scaledMantissa;
}
```

2.5 Mnożenie

Znak wyniku mnożenia wyznaczany jest na podstawie znaków liczb wejściowych. W przypadku, gdy znaki liczb są równe, znak wyniku musi być dodatni (kodowany jako zero). W przeciwnym przypadku znak liczby będącej wynikiem mnożenia jest ujemny (kodowany jako jeden). Wykładnik iloczynu jest sumą wykładników mnożnej i mnożnika, od której należy odjąć obciążenie, ponieważ jest ono uwzględnione w obu liczbach przed wykonaniem działania. Następnie w pętli sumowane są w zmiennej pomocniczej wyznaczone iloczyny częściowe. Po zsumowaniu wszystkich iloczynów częściowych następuje sprawdzenie czy liczba jest dalej znormalizowana. Zostało to wykonane poprzez sprawdzenie czy część całkowita liczby nie przekracza 1. W przypadku, gdy liczba nie jest znormalizowana wykonane zostaje skalowanie wykładnika. Tak uzyskany wynik jest zwracany przez funkcję.

Fragment kodu 8 - implementacja mnożenia

```
ExtendedPrecission ExtendedPrecission::operator*(ExtendedPrecission num)
{
    ExtendedPrecission result(0.0);

    if (this->sign == num.getSign())
        result.sign = 0;
    else
        result.sign = 1;

    result.exponent = this->exponent + num.getExponent() - 16383;

    __uint128_t resultMantissa = 0;
    __uint128_t thisMantissa = this->mantissa;
    uint64_t mask = 0b1;

    resultMantissa += (num.getMantissa() & mask) * this->mantissa;
    for (int i = 1; i < 64; i++)
    {
        resultMantissa += ((num.getMantissa() >> i) & mask)
                          * (thisMantissa << i);
    }

    if ((resultMantissa >> 126) > 1)
        result.exponent += 1;

    result.mantissa = resultMantissa >> 64;

    return result;
}
```

3. Testy jednostkowe

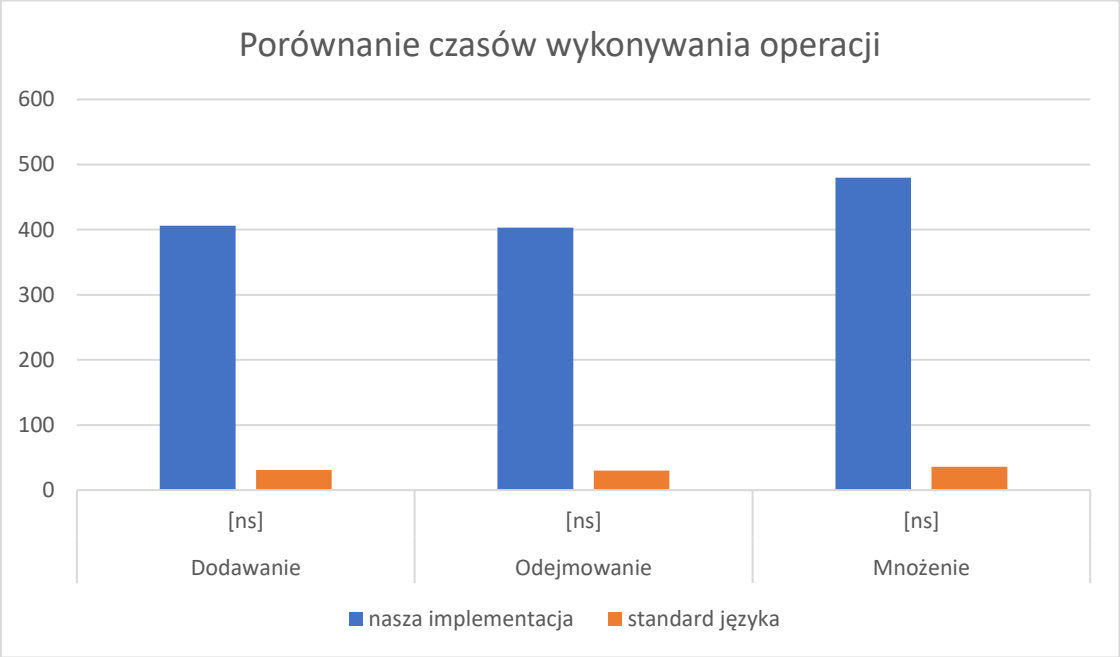
Testy jednostkowe zostały przygotowane w oparciu o bibliotekę Google Test [3]. Sporządzone testy pokrywają wszystkie zaimplementowane mechanizmy: operatory porównania, operator równości, funkcję konwertującą liczby z postaci zmiennoprzecinkowej na format IEEE-754.

Wszystkie sporządzone testy znajdują się w pliku *temp-repo/test/test.cpp*. Aby wykonać testy konieczne jest ich przekompilowanie. W tym celu należy wygenerować plik *Makefile* poprzez wywołanie komendy *cmake CMakeLists.txt* z poziomu terminala. W przypadku braku pakietu *cmake* należy zainstalować go wywołując komendę *sudo apt install cmake*. Konieczne może okazać się także

zainstalowanie dwóch kolejnych pakietów przy użyciu komend *sudo apt install build-essential* oraz *sudo apt install libgtest-dev && cd /usr/src/gtest && sudo cmake CMakeLists.txt && make && sudo cp *.a /usr/lib*. Po wykonaniu tych kroków i pomyślnym wygenerowaniu pliku *Makefile* można przystąpić do przekompilowania testów. W tym celu należy wywołać komendę *make all*, a następnie uruchomić plik *executeTests* w folderze *temp-repo/tests*.

4. Testy wydajnościowe

Przeprowadzone testy wydajnościowe wykazały zgodnie z oczekiwaniami, że nasza implementacja operacji jest znacznie wolniejsza niż ta oferowana przez typ wbudowany w bibliotekę standardową C++. Pomiary czasu wykonywania operacji zostały wykonane 1000000 razy, a wyniki przedstawione na poniższym wykresie zostały uśrednione.



Rysunek 2. Wykres porównawczy czasów wykonywania operacji arytmetycznych

Przy wykorzystaniu narzędzia *gprof* przeprowadzono analizę, aby uzyskać informacje dlaczego wykonana przez nas implementacja jest nieefektywna. Jak można zauważyć, analizując poniższe listingi wygenerowane przez użyte narzędzie profilujące, nasza implementacja wykazuje duże narzuty związane z konwersją liczb, a także dostosowaniem do użytej struktury danych. Takie narzuty operacji powodują znaczny spadek wydajności. Prawdopodobnym rozwiązaniem sytuacji mogłaby okazać się zmiana wewnętrznej reprezentacji liczby.

time	seconds	seconds	calls	Ts/call	Ts/call	name
0.00	0.00	0.00	800	0.00	0.00	ExtendedPrecission::getExponent()
0.00	0.00	0.00	500	0.00	0.00	ExtendedPrecission::getSign()
0.00	0.00	0.00	400	0.00	0.00	ConvertToIeeeExtended(long double, char*)
0.00	0.00	0.00	400	0.00	0.00	ExtendedPrecission::ExtendedPrecission(long double)
0.00	0.00	0.00	200	0.00	0.00	ExtendedPrecission::getMantissa()
0.00	0.00	0.00	200	0.00	0.00	ExtendedPrecission::operator>(ExtendedPrecission)
0.00	0.00	0.00	100	0.00	0.00	carryExponent()
0.00	0.00	0.00	100	0.00	0.00	scaleMantissa(ExtendedPrecission, ExtendedPrecission, OperationType)
0.00	0.00	0.00	100	0.00	0.00	test_add_with_gprof()
0.00	0.00	0.00	100	0.00	0.00	ExtendedPrecission::operator+(ExtendedPrecission)
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I__Z19test_add_with_gprofv
0.00	0.00	0.00	1	0.00	0.00	_GLOBAL__sub_I_carry
0.00	0.00	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)
0.00	0.00	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)

Rysunek 3. Listing uzyskany podczas analizy naszej implementacji programem gprof

5. Wnioski

W projekcie pomyślnie udało się zaimplementować operacje porównywania liczb, oraz działania arytmetyczne z zakresu dodawania, odejmowania oraz mnożenia. Jak pokazały wyniki profilowania, czasy wykonywania operacji są znacznie niż dłuższe niż te wykonywane za pomocą standardowych działań zmiennoprzecinkowych. Prawdopodobnie rozważenie innego formatu przechowywania danych znacznie zmniejszyłoby czas obliczeń związany z konwersjami. Innym sposobem poprawy wydajności byłoby zaimplementowanie wydajniejszych algorytmów obliczeniowych.

Bibliografia

[1] Artykuł w serwisie Wikipedia dotyczący formatu rozszerzonej precyzji - https://en.wikipedia.org/wiki/Extended_precision

[2] Konwersja liczby zmiennoprzecinkowej do formatu rozszerzonej precyzji - http://www.onicos.com/staff/iz/formats/ieee.c

[3] Repozytorium biblioteki do testów jednostkowych Google Test - https://github.com/google/googletest

[4] Artykuł w serwisie Wikibooks dotyczący narzędzia gprof - https://pl.wikibooks.org/wiki/Programowanie_w_systemie_UNIX/gprof