Politechnika Wrocławska
Wydział Elektroniki
Dokumentacja końcowa

Organizacja i Architektura Komputerów: Projekt

Autor:

Aleksander Naporowski 226229
TN 13:15 czwartek

PROWADZĄCY:
dr inż. Tadeusz Tomczak

Temat: Implementacja procedur obliczeń na liczbach zmiennoprzecinkowych za pomocą instrukcji stałoprzecinkowych

19 czerwca 2019

Spis treści

1 Cel projektu

Celem projektu było zaimplementowanie procedur obliczeń na liczbach zmiennoprzecinkowych za pomocą instrukcji stałoprzecinkowych w języku C/C++. Zakres procedur przewidywał operacje dodawania, odejmowania, mnożenia, dzielenia oraz pierwiastkowania liczb 80-bitowych.

2 Liczby zmiennoprzecinkowe

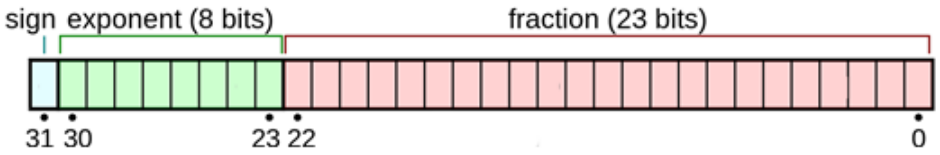
Liczby zmiennoprzecinkowe są komputerową reprezentacją liczb rzeczywistych zapisanych w formie wykładniczej (naukowej). Aby uprościć arytmetykę na tych liczbach, przyjęto ograniczenia zakresu mantysy i wykładnika oraz wprowadzono inne założenia, które reguluje norma IEEE754 (dla liczb zapisanych w kodzie dwójkowym).

$$x = S^{-1} \cdot M \cdot 2^{E}$$

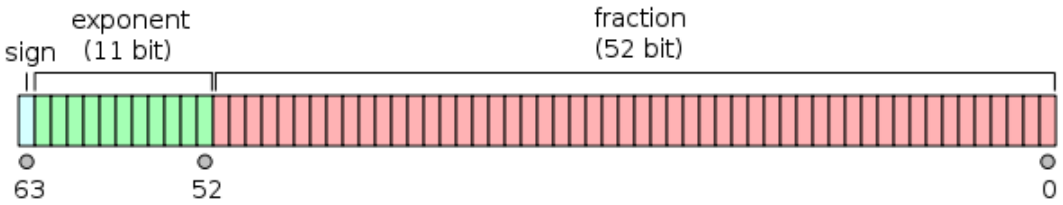
Rysunek 1: Wzór liczby zmienoprzecinkowej

Na rysunku 1 przedstawiony został wzór, dzięki któremu można wyliczyć wartość liczby zmiennoprzecinkowej. We wzorze kolejne zmienne oznaczają: S - znak, M - mantysa, E - wykładnik.

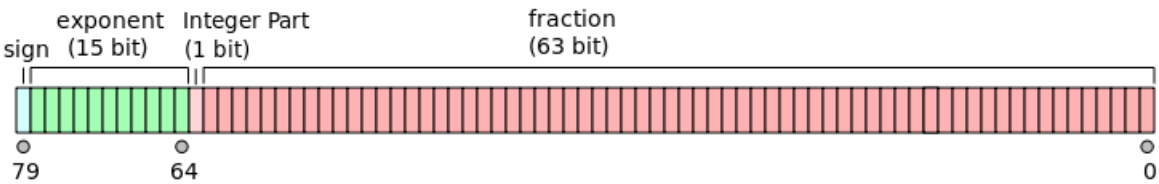
Typy danych zmiennoprzecinkowych różnią się długością, co przekłada się na dokładność prezentowanej liczby. Podstawowymi typami liczb zmiennoprzecinkowych w języku C++ są: *float* (32-bitowa; pojedyncza precyzja), *double* (64-bitowa; podwójna precyzja), *long double* (80-, 96-, 128-bitowa - w zależności od specyfikacji procesora i kompilatora; rozszerzona precyzja).



Rysunek 2: Reprezentacja zmiennoprzecinkowa 32-bitowa



Rysunek 3: Reprezentacja zmiennoprzecinkowa 64-bitowa



Rysunek 4: Reprezentacja zmiennoprzecinkowa 80-bitowa

3 Założenia projektu

Podczas realizacji projektu ustalone zostały następujące założenia:

- Implementacja programu w języku C/C++
- Działania na liczbach zmiennoprzecinkowych: dodawanie, odejmowanie, mnożenie, dzielenie oraz pierwiastkowanie
- Program powinien się kompilować pod systemem operacyjnym Linux
- Zmienna przechowująca liczby zmiennoprzecinkowe ma mieć długość 80 bitów
- Użycie operacji bitowych
- Implementacja testów jednostkowych z użyciem biblioteki Google Test
- Konwersja wejścia jako liczb zapisanych w formacie dziesiętnym do standardu liczb zmiennoprzecinkowych IEEE754

4 Środowisko

Projekt został przygotowany do działania pod kontrolą systemu operacyjnego Linux. W trakcie realizacji projektu przeprowadzone zostały testy programu pod systemem operacyjnym Windows, jednak kod nie działał prawidłowo. Powodem było rzutowanie zmiennej typu *long double* na typ *double*, które jest domyślnie wymuszane pod wyżej wymienionym systemem.

Projekt został przetestowany pod systemem Ubuntu w wersji 18.04.2 LTS 64 bity. Do kompilacji został użyty kompilator GCC (GNU Compiler Collection). IDE, w którym realizowano projekt to Code::Blocks w wersji 16.01.

5 Implementacja

Projekt został zrealizowany z wykorzystaniem systemu operacyjnego Linux w wersji 18.04.2 LTS 64 bity. Kompilator to GCC (GNU Compiler Collection) w wersji 7.4.0.

W celu realizacji projektu, stworzona została unia, dzięki której możliwa była konwersja zmiennej typu *long double* na zmienne składowe stałoprzecinkowe określające znak, wykładnik i mantysę. Użycie 64-bitowego systemu sprawiło, że zmienna typu *long double* miała długość 128 bitów i dzięki zastosowaniu tej unii możliwe było obcięcie zmiennej do 80 bitów.

```
8   typedef union
9   {
10      long double number;
11      struct
12      {
13          uint64_t mantissa : 64;
14          uint16_t exponent : 15;
15          uint16_t sign : 1;
16      }
17      parts;
18  }
19  float80;
```

Rysunek 5: Unia stworzona na cele realizacji projektu

Unia przedstawiona na rysunku 5 składa się ze zmiennej zmiennoprzecinkowej *number* typu *long double* oraz struktury nazwanej *parts*. Struktura, jako ciągły obszar pamięci, pozwala na wydobycie odpowiednich bitów ze zmiennej *number*. W tym celu wykorzystane zostało pojęcie pól bitowych, co daje jeden bit na znak, 15 bitów na wykładnik oraz 64 bity na mantysę. Zastosowane zostały odpowiednie typy zmiennych stałoprzecinkowych.

Podczas realizacji operacji mnożenia potrzebna była zmienna, która przechowa wynik mnożenia mantys obu liczb wejściowych, który może cechować się zwiększoną długością. W tym celu zastosowane zostały rozszerzenia kompilatora GCC dostępne w jego nowszych wersjach i tym samym wykorzystana została zmienna stałoprzecinkowa $\_\_uint128\_t$. Dodatkowym warunkiem użycia tej zmiennej była realizacja projektu w systemie 64-bitowym.

W ustawieniach kompilatora zastosowane zostały następujące flagi dodatkowe:

- *-std=gnu++11* - pozwala na użycie nowszych funkcji języka C++ wprowadzonych przez standaryzację C++11, znana również pod nazwą C++0x
- *-pthread* - flaga dodana na potrzeby Google Test, pozwalająca na wielowątkową pracę programu

W projekcie zostały zrealizowane operacje dodawania, odejmowania i mnożenia, co zostanie opisane bardziej szegółowo w dalszej części rozdziału. Przed wykonaniem operacji uwzględnione zostały szczególne przypadki, np.: NaN, Inf, dwukrotność liczby oraz suma liczb przeciwnych.

W każdej z zaimplementowanych operacji sprawdzane są odpowiednie przypadki przedstawione powyżej.

```
55      //NaN
56      if (!(a_exponent ^ 0x7FFF) && a_mantissa)
57      {
58          result->parts.sign = a->parts.sign;
59          result->parts.exponent = a_exponent;
60          result->parts.mantissa = a_mantissa;
61          return;
62      }
63
64      if (!(b_exponent ^ 0x7FFF) && b_mantissa)
65      {
66          result->parts.sign = b->parts.sign;
67          result->parts.exponent = b_exponent;
68          result->parts.mantissa = b_mantissa;
69          return;
70      }
```

Rysunek 6: Kod sprawdzający poprawność liczby

Operacje dodawania i odejmowania liczb zmiennoprzecinkowych zostały zrealizowane w jednej funkcji. Na początku obliczeń mantysy obu liczb wejściowych wykonywane jest ich przesunięcie bitowe w prawo o jeden bit, aby zostawić miejsce na nadmiar powstały przy ich dodawaniu. Następnie określany jest znak wynikowy oraz przypadki wyjątkowe, które mogą wystąpić podczas dodawania lub odejmowania. Kolejnym krokiem jest sprawdzenie różnicy pomiędzy wykładnikami, co pozwola odpowiednio przesunąć jedną z mantys i wykonać operację sumowania lub odejmowania. Podczas tej operacji wybierany jest także większy wykładnik, który stanowi podstawę wykładnika wynikowego. Ostatnią operacją jest normalizacja mantysy wynikowej oraz w przypadku przepełnienia zwiększenie wykładnika. Ostatecznie, składowe liczby zmiennoprzecinowej są przepisywane do odpowiednich pól w unii, co pozwala na otrzymanie właściwej liczby w formie zmiennoprzecinkowej.

```
153             result_mantissa = a_mantissa + (b_mantissa >> exp_difference);
```

Rysunek 7: Przesuwanie mantysy

Operacja mnożenia rozpoczyna się analogicznie do operacji dodawania i odejmowania. W przypadku mnożenia mantys wykorzystana została zmienna $\_\_uint128\_t$, co pozwoliło uniknąć stracenia nadmiaru powstałego podczas wykonywanej operacji. Uzyskany wynik w formie 128-bitowej jest przesuwany bitowo w lewo do momentu, aż najstarszy bit będzie miał ustawioną wartość. Ostatecznie, taka forma mantysy jest przesuwana w prawo o 64 bity, aby można było zapisać ją w wynikowej zmiennej i $uint64\_t$. Mantysa w tym momencie powinna zostać odpowiednio znormalizowana, jednak realizacja tego kroku nie została ukończona i część wyników może być nieprawidłowa.

```
238     __uint128_t r_mantissa = (__uint128_t)a_mantissa * (__uint128_t)b_mantissa;
239
240     while ((r_mantissa >> 127) == 0)
241     {
242         r_mantissa = r_mantissa << 1;
243     }
244
245     result_mantissa = (uint64_t)(r_mantissa >> 64);
```

Rysunek 8: Mnożenie mantys i wyrównanie wyniku

W opisanych powyżej obliczeniach wykorzytano operatory bitowe:

- *xor* ^ - używany przy sprawdzaniu wartości wykładnika i mantysy
- *przesunięcie w lewo «* - używany przy normalizacji mantysy
- *przesunięcie w prawo »* - używany przy normalizacji mantysy
- *maskowanie* - używane przy sprawdzeniu wartości wykładnika

6 Testy

6.1 Testy jednostkowe

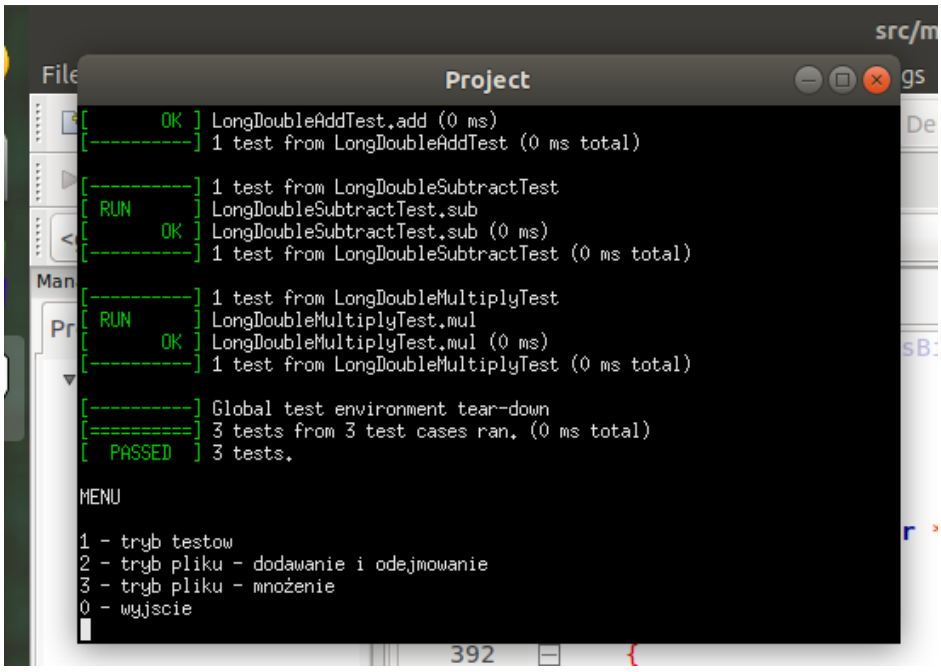
W celu sprawdzenia, czy zaimplementowane procedury na liczbach zmiennoprzecinkowych działają w sposób poprawny, do projektu zostały napisane testy jednostkowe, w oparciu o open-source'owe narzędzie firmy Google - Google Test. W projekcie zaimplementowano szereg testów dla procedur dodawania, odejmowania i mnożenia.

```
463  TEST(LongDoubleMultiplyTest, mul)
464  {
465      ASSERT_DOUBLE_EQ(3888, calculateMultiply(9, 432));
466      ASSERT_DOUBLE_EQ(120, calculateMultiply(60, 2));
467      ASSERT_DOUBLE_EQ(-220, calculateMultiply(10, -22));
468      ASSERT_DOUBLE_EQ(45000, calculateMultiply(-150, -300));
469      ASSERT_DOUBLE_EQ(9.4945806, calculateMultiply(-0.258, -36.8007));
470  }
```

Rysunek 9: Test sprawdzający procedurę mnożenia

Do poprawnego działania biblioteki Google Test zostały dodane opcje w linkerze: *-lgtest*, *-pthread*.

W testach jednostkowych wykorzystana została funkcja porównywania wyników *ASSERT_DOUBLE_EQ*, która pozwala sprawdzić wynik uwzględniając większą precyzję bitową.



Rysunek 10: Wyniki testów jednostkowych przy użyciu biblioteki Google Test

6.2 Testy wydajnościowe

W programie zaimplementowane zostały testy wydajnościowe korzystając z narzędzi dostępnych w przestrzeni nazw *std::chrono*, co przedstawia poniższy kod.

```
21  class Timer {
22  public:
23      std::chrono::high_resolution_clock::time_point startCounting;
24      std::chrono::high_resolution_clock::time_point stopCounting;
25
26      void Start()
27      {
28                  startCounting = std::chrono::high_resolution_clock::now();
29          }
30
31      void Stop()
32      {
33          stopCounting = std::chrono::high_resolution_clock::now();
34          }
35
36      long timeInMS()
37      {
38          return std::chrono::duration_cast<std::chrono::microseconds>(Timer::stopCounting - Timer::startCounting).count();
39          }
40  };
```

Rysunek 11: Klasa *Timer* do pomiarów czasu

Stworzone zostały dwa timery - jeden do pomiaru czasu operacji stałoprzecinkowych, drugi do pomiaru wbudowanych opereracji arytmetycznych na liczbach zmiennoprzecinkowych. Rezultat jest wyświetlany na końcu linii z wynikiem operacji w przypadku testów na danych z pliku, o których mowa niżej.

6.3 Testy na danych z pliku

W programie została dodana możliwość wykonywania zaimplementowanych procedur na liczbach zapisanych w pliku tekstowym *data.txt*. Każda linia pliku zawiera parę liczb oddzielonych od siebie spacjami, które stanowią dane wejściowe dla każdej zaimplementowanej operacji. Liczby zapisane są w postaci dziesiętnej.

Rezultat wyświetlany jest na ekranie w postaci działania wykonanego na wyżej opisanych liczbach. W przypadku, gdy wynik operacji arytmetycznej oraz wynik procedury stałoprzecinkowej są zgodne, na ekranie wyświetla się napis *OK*. Brak takiego napisu świadczy o różnicach, które są spowodowane zaokrągleniem jednego z wyników. Wówczas zamiast *OK* na ekranie wyświetlany jest wynik operacji stałoprzecinkowej.

7 Wnioski

Realizacja projektu pozwoliła na lepsze zrozumienie operacji bitowych w zastosowaniu do liczb zmiennoprzecinkowych. Dzięki temu udało mi się zaimplementować operacje dodawania, odejmowania oraz mnożenia 80-bitowych liczb zmiennoprzecinkowych. Projekt stanowi dla mnie praktyczne wykorzystanie wcześniej poznanej teorii dotyczącej budowy i operacji na liczbach zmiennoprzecinkowych.

Program został zrealizowany w języku C/C++ z wykorzystaniem znacznej ilości operatorów bitowych. Pomocne okazały się informacje i zastosowanie rozszerzeń kompilatora GCC, co ułatwiło pracę nad algorytmem mnożenia.

Wynik testów wydajnościowych obejmujące porównanie czasu obliczeń arytmetycznych i czasu operacji bitowych na wprowadzonych liczbach zmiennoprzecinkowych nie miały praktycznego zastosowania ze względu na zbyt dobrą specyfikację urządzenia, na którym testy te były uruchamiane.

Poznanie i wykorzystanie biblioteki Google Test okazało się bardzo proste. Przy małym nakładzie pracy udało się zaimplementować testy, które objęły wszystkie zrealizowane operacje. Według mnie biblioteka ma spory potencjał, który można wykorzystać w innych projektach, niekoniecznie związanych z tematyką liczb zmiennoprzecinkowych.

Literatura

[1] Operacje bitowe *https://en.cppreference.com/w/cpp/language/operators?fbclid=IwAR2YRoWjFwVSoQHkL1aVkynzJT4rzN$_i$PoSs2gGwnvqy5RvL1MsiRzKPe8M.*

[2] Operacje bitowe w C++ *https://eduinf.waw.pl/inf/alg/002$_s$truct/*0007*.php*

[3] Operacje bitowe *http://www.algorytm.org/kurs-algorytmiki/operacje-bitowe.html*

[4] Extended precision *https://en.wikipedia.org/wiki/Extended$_p$recision*

[5] IEEE *https://eduinf.waw.pl/inf/alg/006$_b$in/*0022*.php*