Organizacja i Architektura Komputerów

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

Dokumentacja projektowa

Wojciech Kur 226078

Prowadzący:
dr inż. Tadeusz Tomczak
Grupa : Czwartek, 11:15

22 czerwca 2019

Spis treści

1 Wstęp

1.1 Cel projektu

Celem projektu jest implementacja biblioteki arytmetyki zmiennoprzecinkowej z wewnętrzną reprezentacją U2, co oznacza, że każda wykonywana operacja oraz przechowywany model FPU powinny występować w tej reprezentacji.

1.2 System uzupełnień do 2 (U2)

Reprezentacja liczby w kodzie uzupełnień do 2 jest obecnie najpopularniejszym sposobem zapisu liczb całkowitych w systemach cyfrowych. Jej popularność wynika z faktu, iż pozwala na wykonywanie operacji dodawania i odejmowania w taki sam sposób jak dla liczb binarnych bez znaku. Charakteryzuje się również istnieniem tylko jednej reprezentacji zera. Wykorzystuje maksymalnie ilość dostępnych bitów, dzięki czemu zakres liczby na n bitach to:

$$[-2^{n-1}, 2^{n-1} - 1] \tag{1}$$

Natomiast wartość można obliczyć za pomocą wzoru:

$$X = (x_k - \beta * \phi(x_k)) * \beta^k + \sum_{i=-m}^{k-1} x_i * \beta^i \tag{2}$$

gdzie β^i - waga pozycji, x_i - cyfra na pozycji oraz $\phi(X))$ - funkcja znaku.

1.2.1 Funkcja znaku

Funkcja znaku określa znak jaki przyjmuje liczba w kodzie uzupełnień do 2 i przedstawia się następująco dla parzystej podstawy systemu:

$$\phi(x) = \begin{cases} 1 & \text{dla } x >= \beta/2 \\ 0 & \text{dla } x < \beta/2 \end{cases} \tag{3}$$

1.2.2 Dodawanie i odejmowanie

Dodawanie oraz odejmowanie wykonujemy tak samo jak dla liczb w kodzie NB. Przeniesienie z ostatniej pozycji może być wykorzystane dla liczb o większej precyzji.

1.2.3 Mnożenie

Mnożenie w U2 różni się od standardowego mnożenia w kodzie NB. Najprostszą metodą jest rozszerzenie dwóch mnożonych liczb dwukrotnie (rozszerzamy do większej liczby, jeśli są różnej długości). Powielamy najstarszy bit każdej z nich aż do uzyskania żądanej długości ciągu. Powielenie najstarszego bitu zapewni nam zgodność znaków oraz w rezultacie całej liczby. Następnie mnożymy liczby pisemnie. Właściwy wynik znajduje się na n bitach, gdzie n to suma długości ciągów mnożonych (przed rozszerzeniem).

Oprócz standardowego algorytmu mnożenia istnieją inne, bardziej wydajne. Pozwalają nie tylko na eliminacje rozszerzeń, dzięki czemu unikamy skalowania mnożonych liczb, ale także zredukowania występowania ilości jedynek w sumach częściowych co znacznie przyspiesza ich dodawanie (algorytm Bootha oraz Bootha-McSorleya).

1.2.4 Dzielenie

Najprostszą metodą dzielenia liczb w U2 jest zapamiętanie znaków i dzielenie modułu dzielnej oraz dzielnika. Po wykonaniu pisemnego dzielenia należy sprawdzić znaki liczb i dostosować odpowiednio znak końcowy. Należy także pamiętać o tym, że reszta z dzielenia (jeśli istnieje) powinna posiadać taki sam znak jak dzielna.

Tak jak w przypadku mnożenia, dzielenie posiada algorytmy pozwalające zredukować ilość wykonywanych operacji. Polegają na wykrywaniu zmiany znaku reszty z dzielenia i na podstawie tego wykonują kolejne kroki (dzielenie odtwarzające oraz dzielenie nieodtwarzające). Umożliwia to dzielenie bez konieczności zamiany liczb na dodatnie oraz ciągłego dostosowania dzielnika do dzielnej.

1.2.5 Nadmiar/Niedomiar

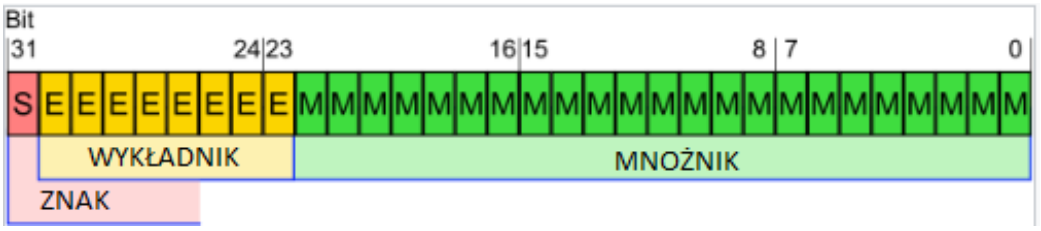
Podczas wykonywania operacji arytmetycznych, wynik może wyjść poza zakres górnej jak i dolnej granicy. Cechą charakterystyczną wystąpienia nadmiaru lub niedomiaru jest nieoczekiwana zmiana znaku. Należy wziąć to pod uwagę i dostosować długość liczby w taki sposób, by prawidłowo ją zapisać.

1.3 Jednostka zmiennoprzecinkowa (standard IEEE)

Liczba zmiennoprzecinkowa to liczba zapisana w formacie:

$$(-1)^S * 2^E * 1, M \tag{4}$$

gdzie S - bit znaku, E - wykładnik oraz M - mnożnik.



Rysunek 1: Reprezentacja zmiennoprzecinkowa IEEE-754 pojedynczej precyzji (Źródło: wikipedia)

Dzięki takiemu podejściu można zapisać bardzo duże jak i bardzo małe liczby na mniejszej liczbie bitów niż byłoby to w przypadku normalnej notacji pozycyjnej.

1.3.1 Dodawanie i odejmowanie

Dodawanie oraz odejmowanie liczb zmiennoprzecinkowych przebiega w kilku etapach. Wymagane jest wyrównanie wykładników. Tylko wtedy możliwe jest dodanie mantys. Następnie mantysy są dodawane, a wynik normalizowany. Przez normalizacje uznaje się sprowadzenie mantysy do postaci „1,M”.

1.3.2 Mnożenie

Na mnożenie jednostki zmiennoprzecinkowej składają się dwie operacje: dodawanie wykładników oraz mnożenie mantys. Po zakończonych operacjach wynik powinien zostać znormalizowany.

1.3.3 Dzielenie

Dzielenie, podobnie jak mnożenie, składa się z dwóch operacji: odejmowania wykładników oraz dzielenia mantys. Tutaj także wymagana jest normalizacja.

2 Koncepcja wykonania projektu

Biorąc pod uwagę warunki jakie powinien spełniać projekt oraz możliwość szybkiej walidacji poprawności wyników, model jednostki zmiennoprzecinkowej będzie wprowadzany za pomocą dwóch wektorów typu INT: pierwszy odpowiada za wartość wykładnika, a drugi za wartość mantysy w reprezentacji U2. Przechowywany będzie natomiast w postaci wektorów typu UINT, a na potrzeby konkretnych działań jego elementy będą rzutowane na wymagany typ. Liczba nie będzie posiadała także wartości specjalnych ze względu na specyfikacje.

2.1 Uproszczenie zapisu

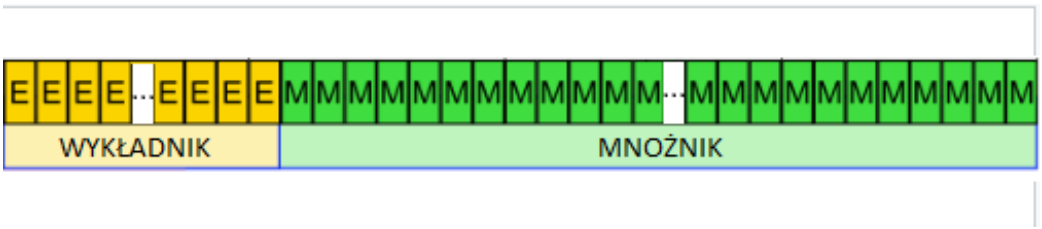
Jednostka zmiennoprzecinkowa została uproszczona do postaci:

$$FloatingPoint(wykadnik(u2), mantysa(u2)) \tag{5}$$

Ze względu na system U2, wykładnik nie posiada obciążenia. W celu ułatwienia obliczeń mantysa jest liczbą całkowitą, a normalizacja polega na usunięciu zbędnych (o najmłodszych pozycjach) zer z ciągu. Zatem format liczby zmiennoprzecinkowej będzie przedstawiał się następująco:

$$2^E * M \tag{6}$$

gdzie E - wykładnik w systemie U2, M - mnożnik jako liczba całkowita U2



Rysunek 2: Reprezentacja zmiennoprzecinkowa dowolnej precyzji z wewnętrzną reprezentacją U2

2.2 Wprowadzanie liczby

Ciąg wejściowy wprowadzany jest w formacie BIG-ENDIAN (najstarszy element jako pierwszy). Obliczenia wykonywane są w formacie LITTLE-ENDIAN (najmłodszy element jako pierwszy).

2.3 Drukowanie liczby

Liczbę można wydrukować na standardowe wyjście za pomocą przeciążonego operatora. Liczba podawana jest w reprezentacji U2 w formacie BIG-ENDIAN.

3 Implementacja liczby zmiennoprzecinkowej

3.1 Struktura

Kod źródłowy 1: Deklaracja struktury jednostki zmiennoprzecinkowej

```
1  private:
2      std::vector<uint32_t> exponent_;
3      std::vector<uint32_t> mantissa_;
4  public:
5      /**
6      * Standard constructors
7      */
8      explicit FloatingPoint(
9          const std::vector<int32_t>& exponent,
10         const std::vector<int32_t>& mantissa);
```

Kod źródłowy 2: Implementacja struktury jednostki zmiennoprzecinkowej

```
1  FloatingPoint::FloatingPoint(
2      const std::vector<int32_t>& exponent,
3      const std::vector<int32_t>& mantissa) {
4
5      exponent_ =
6          std::vector<uint32_t>(exponent.begin(),
7                                exponent.end());
8
9      mantissa_ =
10         std::vector<uint32_t>(mantissa.begin(),
11                               mantissa.end());
12
13     std::reverse(exponent_.begin(), exponent_.end());
14     std::reverse(mantissa_.begin(), mantissa_.end());
15 }
```

3.2 Dodawanie

Przeciążony operator dodawania składa się z:

- Sprawdzenia kompatybilności dwóch liczb - warunkiem koniecznym są równe długości wykładników oraz mantys. Jeśli warunek nie jest spełniony, dostosowuje liczbę mniejszą do większej pod względem długości wektorów.
- Dostosowania mniejszego wykładnika do większego. Następuje inkrementacja oraz odpowiednie przesunięcie mantysy w prawo (dzielenie przez 2 - wektor powiększany jest o 0 lub 1 przed najstarszym bitem zależnie od znaku)
- Ponownego sprawdzenia kompatybilności i dostosowania do nowej liczby
- Dodania mantys
- Normalizacji - przesunięcia mantysy w lewo (mnożenia przez 2) aż do uzyskania najmniejszej liczby całkowitej (bez utraty dokładności) oraz odpowiedniej dekrementacji wykładnika

Kod źródłowy 3: Dodawanie liczb zmiennoprzecinkowych

```
1  FloatingPoint operator+(
2      const FloatingPoint &x,
3      const FloatingPoint &y) {
4
5      std::vector<FloatingPoint> newArgs =
6          FloatingPoint::compatibility(x, y);
7
8      std::vector<uint32_t> exponent;
9      std::vector<uint32_t> mantissa;
10
11     // Dodawanie
12     std::vector<uint32_t> rest =
13         FloatingPoint::alignExponents(newArgs[0], newArgs[1]);
14
15     newArgs = FloatingPoint::compatibility(newArgs[0],
16                                            newArgs[1], true);
```

```
17
18      exponent = newArgs[0].exponent_;
19
20      mantissa =
21          Utils::addVectors(newArgs[0].mantissa_,
22                                  newArgs[1].mantissa_);
23
24      // Normalizacja
25      if (!rest.empty()) {
26          bool flag = false;
27          for (int i = 0; i < rest.size(); i ++) {
28              if (rest[i] != 0) flag = true;
29          }
30          if (flag) {
31              FloatingPoint::shiftLeft(mantissa, rest);
32              uint32_t subSize =
33                  exponent.size() - rest.size();
34
35              for (int i = 0; i < subSize; i ++) {
36                  rest.push_back(0);
37              }
38              rest = FloatingPoint::negate(rest);
39
40              exponent = Utils::addVectors(exponent, rest);
41          }
42      }
43
44      while (mantissa[0] == 0)
45          mantissa.erase(mantissa.begin());
46
47      std::reverse(exponent.begin(), exponent.end());
48      std::reverse(mantissa.begin(), mantissa.end());
49
50      return FloatingPoint(
51      std::vector<int32_t>(exponent.begin(), exponent.end()),
52      std::vector<int32_t>(mantissa.begin(), mantissa.end())
53      );
54  }
```

3.3 Odejmowanie

Implementacja odejmowania polega na zanegowaniu drugiej liczby, a następnie dodaniu ich do siebie.

Kod źródłowy 4: Odejmowanie liczb zmiennoprzecinkowych

```
1 FloatingPoint operator-(const FloatingPoint &x,
2     const FloatingPoint &y) {
3
4     FloatingPoint newY = y;
5     newY.mantissa_ =
6         FloatingPoint::negate(newY.mantissa_);
7
8     return x + newY;
9 }
```

3.4 Mnożenie

Operator mnożenia składa się z:

- Sprawdzenia kompatybilności wprowadzonych liczb i dostosowanie do właściwego formatu
- Dodania wykładników
- Wymnożenia mantys
- Normalizacji - przesunięcia mantysy w prawo (dzielenia przez 2) aż do uzyskania najmniejszej liczby całkowitej (bez utraty dokładności) oraz odpowiedniej dekrementacji wykładnika

Kod źródłowy 5: Mnożenie liczb zmiennoprzecinkowych

```
1 FloatingPoint operator*(const FloatingPoint &x,
2     const FloatingPoint &y) {
3     std::vector<FloatingPoint> newArgs =
4         FloatingPoint::compatibility(x, y);
5
6     std::vector<uint32_t>
7         exponent(newArgs[0].exponent_.size(), 0);
```

```
 8       std::vector<uint32_t>
 9           mantissa(newArgs[0].mantissa_.size(), 0);
10
11       // Mnozenie
12       exponent = Utils::addVectors(newArgs[0].exponent_,
13                                    newArgs[1].exponent_);
14
15       mantissa = Utils::mulVectors(newArgs[0].mantissa_,
16                                    newArgs[1].mantissa_);
17
18       std::vector<uint32_t> counter(mantissa.size(), 0);
19
20       // Normalizacja
21       for (int i = 0; i < mantissa.size(); i ++) {
22           if (mantissa[i] == 0) {
23               Utils::addBig(counter, 32);
24           }
25           else {
26               std::bitset<32> checkPosition(mantissa[i]);
27               for (int j = 0; j < 32; j ++) {
28                   if (checkPosition[j] == 1) break;
29                   Utils::addBig(counter, 1);
30               }
31               break;
32           }
33       }
34
35       for (int i = counter.size() - 1; i >= 0; i --) {
36           if (counter[i] != 0) {
37               auto shift =
38                   FloatingPoint::shiftRight(mantissa, counter);
39               break;
40           }
41       }
42
43       exponent = Utils::addVectors(exponent, counter);
44
45       while (((mantissa.back() >> 31) & 1) ==
```

```
46                  ((mantissa[mantissa.size() - 2] >> 31) & 1)) {
47
48              if (mantissa.back() == 0 ||
49                  (int32_t)mantissa.back() == -1)
50                      mantissa.pop_back();
51
52              else break;
53          }
54
55          std::reverse(mantissa.begin(), mantissa.end());
56          std::reverse(exponent.begin(), exponent.end());
57
58          return FloatingPoint(
59              std::vector<int32_t>(exponent.begin(), exponent.end()),
60              std::vector<int32_t>(mantissa.begin(), mantissa.end())
61          );
62      }
```

3.5 Dzielenie

Implementacja dzielenia składałaby się z:

- Sprawdzenia kompatybilności i dostosowania liczb
- Odjęcia wykładników
- Zamiany ujemnych mantys na dodatnie
- Dzielenia pisemnego mantys aż do uzyskania danej precyzji
- Określenia znaku wyniku
- Normalizacji wyniku

Niestety dzielenia nie udało się zaimplementować.

4 Plan wykonania testów

Zostaną przeprowadzone trzy rodzaje testów:

- Testy jednostkowe w celu sprawdzenia poprawności otrzymanych wyników. Ponadto testy są częścią metodyki TDD (Test driven development), więc zostaną napisane przed wykonaniem danego algorytmu.
- Testy wydajnościowe są podzielone na trzy części i wskażą złożoność czasową algorytmów. Dzięki temu będzie można oszacować limity biblioteki i dla jakich wartości można ją stosować.
- Test wykorzystywanych mechanizmów procesora ma sprawdzić czy biblioteka jest wspomagana natywnie przez procesor.

Dodatkowo kod zostanie skompilowany z flagą „-O3" co oznacza 3 stopień optymalizacji rozmiaru kodu oraz czasu wykonywania - kosztem dłuższego czasu kompilacji.

4.1 Testy jednostkowe

Uwzględnione zostały testy znaku wykładnika oraz mantysy, kompatybilności dwóch liczb, poprawnego działania dodawania (w tym odejmowania), poprawnego działania mnożenia oraz poprawność działań dla dużych liczb.

4.1.1 Test znaku

Kod źródłowy 6: Przykładowy test znaku

```
1  TEST(MantissaSignTest, Positive) {
2      FloatingPoint fp = FloatingPoint({0}, {1});
3      EXPECT_EQ(0, fp.getSign());
4      fp = FloatingPoint({0}, {INT32_MAX});
5      EXPECT_EQ(0, fp.getSign());
6      fp = FloatingPoint({0}, {0});
7      EXPECT_EQ(0, fp.getSign());
8      fp = FloatingPoint({0}, {INT32_MAX, -1});
9      EXPECT_EQ(0, fp.getSign());
10 }
```

4.1.2 Test kompatybilności

Kod źródłowy 7: Przykładowy test kompatybilności

```
1 TEST(CompatibilityTest, 2Elements) {
2 FloatingPoint fp1 = FloatingPoint({-1}, {-1});
3 FloatingPoint fp2 = FloatingPoint({0, 2}, {1});
4 FloatingPoint result = fp1 + fp2;
5
6 EXPECT_EQ(result.getExponent().size(), 2);
7 }
```

4.1.3 Test dodawania

Kod źródłowy 8: Przykładowy test dodawania

```
1  TEST(AdderHugeNumberTest, PositiveFP) {
2      FloatingPoint fp1 = FloatingPoint({1, 65}, {1, -1});
3      FloatingPoint fp2 = FloatingPoint({1, 1}, {-1, 1});
4      FloatingPoint result = fp1 + fp2;
5
6      ASSERT_THAT(result.getMantissa(true),
7          testing::ElementsAre(1, -2, -1, 1));
8
9      ASSERT_THAT(result.getExponent(true),
10         testing::ElementsAre(1, 1));
11 }
```

4.1.4 Test mnożenia

Kod źródłowy 9: Przykładowy test mnożenia

```
1 TEST(MultiplierTest, MoreElements) {
2     FloatingPoint fp1 = FloatingPoint({1, 5}, {-1, INT32_MAX});
3     FloatingPoint fp2 = FloatingPoint({5, 3}, {INT32_MAX, -1});
4
5     FloatingPoint result = fp1 * fp2;
6
7     ASSERT_THAT(result.getExponent(true),
8         testing::ElementsAre(6, 8));
9
```

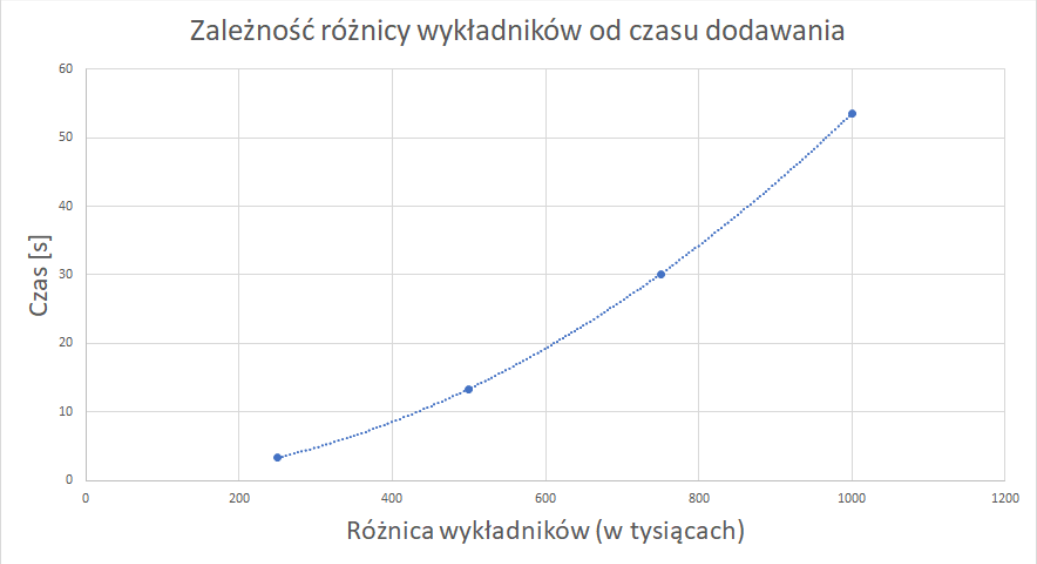
```
10      ASSERT_THAT(result.getMantissa(true),
11          testing::ElementsAre(3221225472, 2147483648, 2147483649));
12  }
```

4.2 Testy wydajnościowe

Każdy z testów wydajnościowych zostanie przeprowadzony 4 razy, a wynik uśredniony.

4.2.1 Test zależności różnicy wykładników od czasu dodawania

Test został przeprowadzony dla różnicy wykładników dwóch liczb rzędu 250 tysięcy, 500 tysięcy, 750 tysięcy oraz 1 miliona. W przypadku ostatniego testu oznacza to dość dużą różnicę między liczbami, biorąc pod uwagę brak utraty dokładności (mantysa rozszerzana w nieskończoność).



Rysunek 3: Zależność różnicy wykładników od czasu dodawania

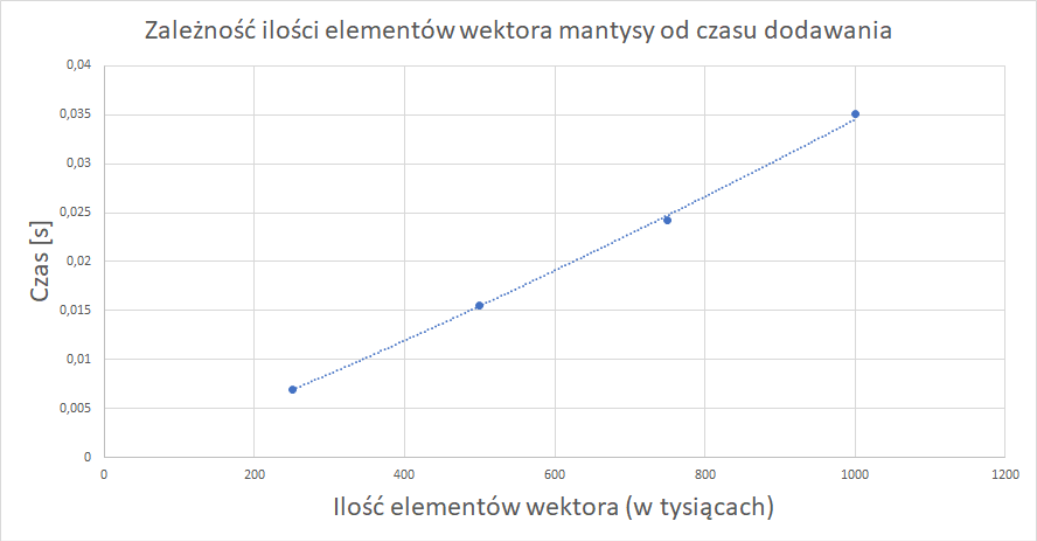
Wykres (Rys. 3) przypomina funkcję potęgową. Operacją najbardziej zajmującą obliczeniowo jest wyrównanie wykładników. Metody „shiftRight" oraz „shiftLeft" powodują, iż złożoność obliczeniowa dodawania to:

$$O(n * m) \tag{7}$$

gdzie n - wielkość wektora wyjściowego mantysy oraz m - różnica wykładników.

4.2.2 Test zależności ilości elementów wektora mantysy od czasu dodawania

Test został przeprowadzony dla 250 tysięcy, 500 tysięcy, 750 tysięcy oraz 1 miliona elementów wektora mantysy. Dodawanie elementów wektora jest wspierane natywnie przez procesor. Przeniesienie między jego elementami nie wykorzystuje flagi „carry" - jest dodatkową operacją wymagającą osobnej zmiennej.



Rysunek 4: Zależność ilości elementów mantysy od czasu dodawania

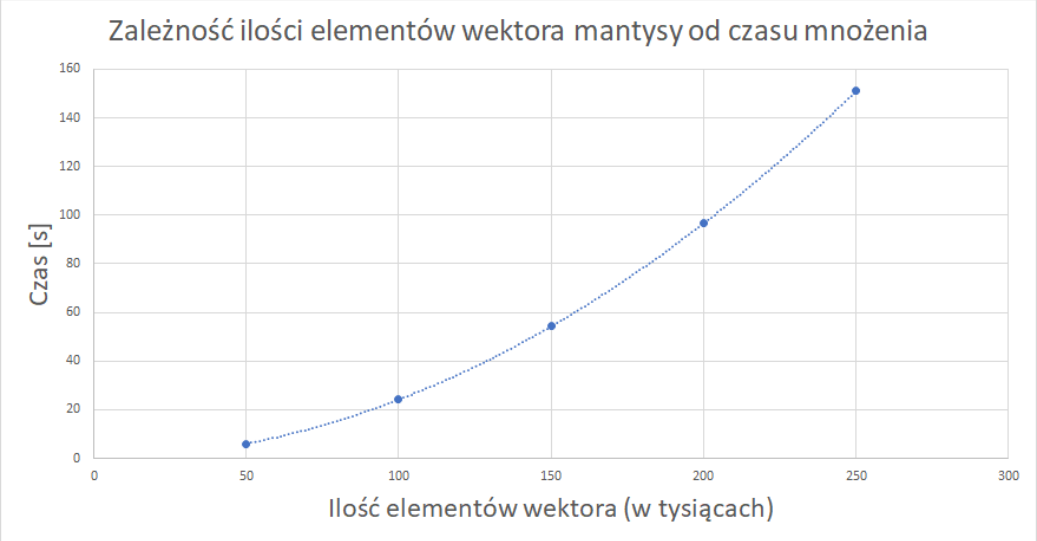
Przyrost czasu w zależności od ilości elementów mantysy jest niemalże liniowy. Dla pomijalnych różnic wykładników, dodawanie wektorów jest zależne jedynie od ich wielkości. Z analizy kodu jak i wykresu (Rys. 4) wynika, iż metoda „addVectors" ma złożoność obliczeniową:

$$O(n) \tag{8}$$

gdzie n - wielkość wektora mantysy.

4.2.3 Test zależności ilości elementów wektora mantysy od czasu mnożenia

Test został przeprowadzony dla 50 tysięcy, 100 tysięcy, 150 tysięcy, 200 tysięcy oraz 250 tysięcy elementów wektora mantysy. Mnożenie jest wspierane natywnie przez procesor, jednakże wymaga dodatkowej operacji dodawania przy przeniesieniu połowy bitów wyniku na starszą pozycję (element wektora).



Rysunek 5: Zależność ilości elementów mantysy od czasu mnożenia

Przyrost czasu w zależności od ilości elementów wektora mantysy jest funkcją potęgową. Wynika to z faktu konieczności mnożenia każdego elementu wektora mnożnika przez wektor mnożnej. Potwierdza to wykres (Rys. 5). Z analizy kodu wynika natomiast, że złożonośc obliczeniowa wynosi:

$$O(n^2) \tag{9}$$

gdzie n - wielkość wektora mantysy.

4.3 Test wykorzystywanych mechanizmów procesora

4.3.1 Mechanizm dodawania

Sprawdzenie wykorzystania mechanizmów dodawania zostało wykonane poprzez disassemble fragmentu metody odpowiedzialnej za dodawanie wektorów (addVectors)

Kod źródłowy 10: Sprawdzenie mechanizmu dodawania

```
1  mov   (%r12),%r11
2  xor   %edx,%edx
3  xor   %ecx,%ecx
4  nop
5  mov   (%rdi,%rdx,1),%eax
6  mov   (%r11,%rdx,1),%r13d
7  add   %r13,%rax
8  add   %rcx,%rax
9  mov   %eax,(%r8,%rdx,1)
10 shr   $0x20,%rax
11 add   $0x4,%rdx
12 mov   %eax,%ecx
13 mov   %eax,0x4(%rsp)
14 cmp   %rdx,%r9
```

Można zauważyć, iż w procesie dodawania zostały wykorzystane rozkazy „add". Ich wykorzystanie polegało na dodawaniu elementów uint wektora poprzez operator „+".

4.3.2 Mechanizm mnożenia

Sprawdzenie wykorzystania mechanizmow mnożenia zostało wykonane poprzez disassemble fragmentu metody mnożenia wektorów (mulVectors)

Kod źródłowy 11: Sprawdzenie mechanizmu mnożenia

```
1 mov    0x3c(%rsp),%eax
2 mov    (%rsi,%rbx,4),%ebx
3 lea    0x0(%rbp,%rax,1),%ecx
4 movslq %ecx,%rax
5 add    $0x1,%ecx
```

```
 6  lea     (%r9,%rax,4),%r11
 7  mov     (%rdx),%eax
 8  movslq  %ecx,%rcx
 9  imul    %rbx,%rax
10  add     %rbx,%rax
11  mov     0x40(%rsp),%rbx
12  mov     %eax,(%r11)
13  shr     $0x20,%rax
14  mov     %eax,(%r9,%rcx,4)
15  cmp     %rbx,%r10
```

Tak samo jak w przypadku dodawania, możemy zauważyć rozkaz „add" oraz nowy „imul". Rozkaz „imul" to mnożenie z uwzględnieniem znaku co informuje nas, że procesor mimo mnożenia elementów bez znaku wykorzystuje inny rozkaz, być może ze względów optymalizacyjnych. Rozkaz „add" jest widoczny ze względu na konieczność dodawania starszej części bitów wyniku mnożenia na kolejną pozycję wektora.

5 Podsumowanie

5.1 Wnioski

Dodawanie jak i odejmowanie liczb zmiennoprzecinkowych dowolnej precyzji jest dość wymagające obliczeniowo dla liczb znacznie różniących się od siebie. Jeśli liczby dowolnej wielkości będą do siebie zbliżone (250 tysięcy różnicy wykładnika), biblioteka powinna natychmiast podać wynik działania.

Mnożenie o dużej dokładności mantysy wykonuje się nieco wolniej od ich dodawania, jednakże dla większej różnicy wykładników nie ma to znaczenia, gdyż są one dodawane. Nie jest wymagane ich skalowanie, przez co działanie to jest o wiele mniej złożone.

Dzielenie, choć niezrealizowane, wydaje się najbardziej złożoną obliczeniowo operacją. Wymagane jest wielokrotne skalowanie dzielnej oraz wykonanie odpowiednich przeniesień podczas dzielenia elementów wektora.

5.2 Napotkane trudności

Największą trudnością był sam koncept liczby zmiennoprzecinkowej o nieskończonej precyzji oraz z wewnętrzną reprezentacją U2 ze względu na trudności w znalezieniu podobnych bibliotek. Na szczęście po zgłębieniu specyfikacji reprezentacji U2 udało się wykonać większą część standardowych operacji.

Nie udało się wykonać dzielenia ze względu na trudności wynikające z konieczności ciągłej zmiany wielkości dzielnej, co w połączeniu z reprezentacją mantysy opartej na wektorze generowało zbyt wiele błędów.

Lista wzorów

Lista kodów źródłowych

Literatura

[1] *http://zak.ict.pwr.wroc.pl/materials/Arytmetyka%20komputerow/* - *Materiały z kursu Arytmetyka komputerów. (Odwiedzono: 22 czerwca 2019)*

[2] *https://eduinf.waw.pl/inf/alg/006_bin/0018.php* - *Zapis oraz operacje w systemie uzupełnień do 2. (Odwiedzono: 22 czerwca 2019)*

[3] *https://eduinf.waw.pl/inf/alg/006_bin/0008.php* - *Zapis oraz operacje w formacie zmiennoprzecinkowym (Odwiedzono: 22 czerwca 2019)*

[4] *https://pl.wikipedia.org/wiki/IEEE_754* - *Format zmiennoprzecinkowy (Odwiedzono: 22 czerwca 2019)*