Architektura komputerów 2 – projekt

Temat: Działania na liczbach dużego rozmiaru (> 1024 bity, mnożenie, dzielenie)

**Skład grupy:**

Hubert Olkiewicz 200145

Paweł Janikowski 209878

**Termin zajęć:** Wtorek TP 9.15

**Prowadzący:** Dr hab. inż. Janusz Biernat, prof. nadzw. PWr

Spis treści

[**1.** **Opis tematu** 3](#_Toc453276279)

[**2.** **Założenia i ograniczenia** 4](#_Toc453276280)

[**2.1.** **Środowisko pracy** 4](#_Toc453276281)

[**2.2.** **Format wejściowy** 4](#_Toc453276282)

[**2.3.** **Format wyjściowy** 4](#_Toc453276283)

[**2.4.** **Własna implementacja zmiennoprzecinkowa** 4](#_Toc453276284)

[**2.4.1.** **Reprezentacja** 4](#_Toc453276285)

[**2.4.2.** **Mantysa** 5](#_Toc453276286)

[**2.4.3.** **Eksponenta** 5](#_Toc453276287)

[**2.5.** **Wyjątki** 5](#_Toc453276288)

[**3.** **Opis rozwiązania** 6](#_Toc453276289)

[**3.1.** **Przygotowanie eksponenty do konwersji** 6](#_Toc453276290)

[**3.2.** **Konwersja do reprezentacji binarnej i umieszczenie w tablicach** 7](#_Toc453276291)

[**3.3.** **Skorygowanie mantysy** 7](#_Toc453276292)

[**3.4.** **Dodawanie tablicowe** 9](#_Toc453276293)

[**3.5.** **Odejmowanie tablicowe** 9](#_Toc453276294)

[**3.6.** **Mnożenie** 9](#_Toc453276295)

[**3.7.** **Dzielenie** 12](#_Toc453276296)

[**3.8.** **Wyświetlenie wyniku w postaci szesnastkowej** 14](#_Toc453276297)

[**4.** **Działanie programu - testy** 16](#_Toc453276298)

[**4.1.** **Mnożenie** 16](#_Toc453276299)

[**4.2.** **Dzielenie** 17](#_Toc453276300)

[**4.3.** **Sprawdzanie poprawności** 18](#_Toc453276301)

[**4.4.** **Pomiary czasu** 18](#_Toc453276302)

[**5.** **Wnioski** 20](#_Toc453276303)

[**6.** **Literatura** 20](#_Toc453276304)

# Opis tematu

Tematem projektu była implementacja mnożenia i dzielenia na argumentach dużego rozmiaru (liczbach 1024 bitowych i większych) oraz własna implementacja zmiennoprzecinkowa. Po kolei zostały realizowane następujące etapy pracy:

* Wczytywanie oraz konwersja liczby do tablic,
* Implementacja algorytmu mnożenia,
* Rozszerzenie algorytmu mnożenia na działanie wielotablicowe,
* Implementacja algorytmu dzielenia,
* Rozszerzenie algorytmu dzielenia na mnożenie wielotablicowe,
* Dodanie własnej implementacji zmiennoprzecinkowej,
* Wczytywanie danych wejściowych i wyświetlanie danych wyjściowych w systemie szesnastkowym.

1. Założenia i ograniczenia
   1. **Środowisko pracy**

Projekt został zaimplementowany w języku C++ (w standardzie C++11) z wykorzystaniem wstawek asemblerowych przy niektórych obliczeniach, w celu ich przyspieszenia. Do kompilacji programu skorzystano z kompilatora MinGW (port GCC z systemu Linux na Windows).

* 1. **Format wejściowy**

Podawane są dwie liczby, na których zostaną wykonane operacje mnożenia i dzielenia. Argumenty wejściowe dla jednej liczby podawane są w następującej kolejności:

1. Ciąg znaków określający wartość liczbową mantysy,
2. Pojedynczy znak ‘*+*’ lub ‘*-*‘ – dla mantysy,
3. Ciąg znaków określający wartość eksponenty,
4. Pojedynczy znak ‘*+*’ lub ‘*-*‘ dla eksponenty.

Znaki ‘*+*’ (plus) i ‘*-*‘ (minus) podawane są w postaci jednego znaku typu *char* i oznaczają, że podana wartość liczby jest dodatnia lub ujemna. Ciągi znaków dla wartości mantysy i eksponenty są typu *string*. Użytkownik podając dane wejściowe, używa znaków dostępnych dla systemu szesnastkowego tj. cyfr od *0 – 9* oraz liter od *a – f* lub *A – F*.

* 1. **Format wyjściowy**

Dane wyjściowe wypisywane są na ekranie w konsoli w postaci znormalizowanej, w systemie szesnastkowym. Są to wczytane na wejściu liczby oraz wyniki mnożenia i dzielenia tych liczb przez siebie.

* 1. **Własna implementacja zmiennoprzecinkowa**
     1. **Reprezentacja**

Po wczytaniu danych wejściowych, liczba przyjmuje następującą postać:

.

Podstawa wynosi 16 a wartości mantysy i eksponenty to dane wejściowe programu. Zanim to jednak nastąpi, wczytany ciąg znaków typu *string* poddany jest konwersji do reprezentacji binarnej.

Reprezentacja binarna liczby, na której wykonywane są działania mnożenia i dzielenia, przechowywana jest w tablicach 32-bitowych typu *unsigned int*. Gdy nie ma miejsca w jednej tablicy, część liczby jest „odcinana” i umieszczana w kolejnej.

* + 1. **Mantysa**

Może przyjmować wartości całkowite jak i zawierające część ułamkową, którą można podać po przecinku lub kropce. Może być ujemna lub dodatnia.

* + 1. **Eksponenta**

Może przyjmować wartości całkowite i ułamkowe podobnie jak w przypadku mantysy. Może być również dodatnia lub ujemna.

* 1. **Wyjątki**

Dzielenie przez ‘0’ - program wyświetla stosowny komunikat, a wynik dzielenia wynosi 0. Dzielenie mniejszej liczby przez większą - algorytm dzielenia został zaimplementowany jedynie w wersji całkowitoliczbowej. Stosowny komunikat o przekroczeniu precyzji zostanie wyświetlony, a wynik dzielenia wynosi 0.

1. Opis rozwiązania

W niniejszym rozdziale przedstawiono podstawowe metody wykorzystywane w programie. Odpowiadają one za prawidłowe działanie i są wykorzystywane podczas każdego użycia programu. Ich nazwy pochodzące z kodu źródłowego oznaczono kursywą i zakończono dwoma nawiasami okrągłymi.

* 1. Przygotowanie eksponenty do konwersji

*Count\_Exponent\_And\_Normalize()* – funkcja jako argument przyjmuje wczytaną mantysę w formie *string* i odpowiada za dostosowanie jej do konwersji i wyliczenie długości przecinka w liczbę. Wykonuje ona kilka czynności. Sprawdza, czy wprowadzono przecinek bądź kropkę, po czym następuje wyszukanie jego pozycji i usunięcie. Sprawdza czy nie wprowadzono niepotrzebnych 0 na początku liczby, które jeśli istnieją, są obcinane. Zwraca ilość miejsc po przecinku (w notacji dziesiętnej), również typu *string*, z dodanym na początku znakiem ‘+’ lub ‘-‘, który ustalany jest na podstawie czy, aby znormalizować liczbę należy przesunąć przecinek w lewo (‘+’) czy w prawo (‘-’). Po usunięciu przecinka zakładamy, że mantysa jest znormalizowana (tj. w postaci x,xxx…, gdzie x to cyfra).

string count\_Exponent\_And\_Normalize(string &numberString)

{

stringstream ss;

size\_t found;

string buffor=numberString.substr(0,2);

if(!(buffor=="0," || buffor=="0

{

buffor="+";

found = numberString.find('.');

if (found != string::npos)

{

numberString.erase(numberString.begin()+found);

}

else

{

found = numberString.find(',');

if(found != string::npos)

{

numberString.erase(numberString.begin()+found);

}

else

{

found = numberString.size();

}

}

}

else

{

buffor="-";

found = numberString.find\_first\_of("123456789abcedfABCDEF");

numberString=numberString.substr(found,numberString.size()-found);

}

ss << found-1;

buffor.append(ss.str());

return buffor;

}

Listing1. Kod źródłowy funkcji *count\_Exponent\_And\_Normalize()*.

* 1. Konwersja do reprezentacji binarnej i umieszczenie w tablicach

*convertToBinArrayFromHex()* to funkcja przeprowadzająca konwersję z szesnastkowo wczytanej liczby, zarówno mantysy jak i eksponenty, na reprezerntację binarną, podzieloną na odpowiednią ilość tablic. Jako argument przyjmuje liczbę w postaci *string* a zwraca jej wartość binarną w postaci tablic *unsigned int*. Odbywa się to poprzez przestawianie bitów, czyli shitf, w lewą stronę o 4 pozycje dla każdego znaku.

int convertToBinArrayFromHEX(string toConvert, vector<unsigned int> &number)

{

number.push\_back(0);

for(int i =0; i<=toConvert.size()-1;i++)

{

for(int j =0; j<4;j++)

shiftArrayLeft(number,false);

number.at(0)+=changeHexDigitToDec(toConvert.at(i));

}

}

Listing 2. Kod źródłowy funkcji *convertToBinArrayFromHEX()*.

* 1. Skorygowanie mantysy

Po mantysie i eksponencie, przeprowadzana jest jeszcze konwersja liczby zwróconej z funkcji *Count\_Exponent\_And\_Normalize()* (wychwycona ilość przesunięc przecinka potrzebna do normalizacji mantysy) poprzez funkcję *convertToBinArrayFromDEC()*. Funkcja ta konwertuje liczbę dziesiętna zapisaną w postaci *string* do tablicy *unsigned int* która odpowiada reprezentacji binarnej liczby. Wykorzystana tutaj funkcja wbudowana *\_\_builtin\_uadd\_overflow()* pozwala na wykrycie nadmiaru i odpowiednie skorygowanie liczby. Algorytm jest wykonywany za pomocą schematu Hornera, i wykorzystuje wstawkę asemblerową do mnożenia przez 2.

int convertToBinArrayFromDEC(string toConvert, vector<unsigned int> &number){

vector<unsigned int> buffor;

char \* digits = new char [toConvert.length()+1];

strcpy (digits, toConvert.c\_str());

int digit;

for(int i=0; i<toConvert.length();i++)

{

if(i==0)

{

number.push\_back((int)digits[i]-'0');

}

else

{

digit = (int)digits[i]-'0';

for(int j=0; j<number.size();j++)

{

if(\_\_builtin\_uadd\_overflow(number.at(j), digit,&number.at(j)))

{

digit=1;

if(j+1==number.size())

{

number.push\_back(1);

break;

}

else

continue;

}

else

{

break;

}

}

}

vector<unsigned int> buffor;

unsigned int lower,higher;

for(int j=0; j<number.size() && i<toConvert.length()-1; j++)

{

asm ("movl %2, %%eax;"

"mull %3;"

"movl %%eax, %0;"

"movl %%edx, %1;"

: "=r" ( lower ), "=r" ( higher )

: "r" ( number.at(j) ), "r" ( 10 )

:"eax","edx"

);

buffor.push\_back(higher

number.at(j)=lower;

}

if(number.size()==buffor.size())

number.push\_back(0);

for(int j=0; j<buffor.size(); j++)

{

digit=buffor.at(j);

for(int z=j+1; z<number.size(); z++)

{

if(\_\_builtin\_uadd\_overflow(number.at(z), digit,&number.at(z)))

{

digit=1;

if(z==number.size())

{

number.push\_back(1);

break;

}

else

continue;

}

else

{

break;

}

}

}

if(number.back()==0)

number.pop\_back();

}

}

Listing 3. Kod źródłowy metody *convertToBinArrayFromDec()*.

* 1. Dodawanie tablicowe

Wykorzystywana przy mnożeniu funkcja *addArrays()*,wykonuje dodawanie binarne dwóch tablic otrzymanych w parametrach, z uwzględnieniem i korektą nadmiaru. Wykorzystuje ona sposób “pisemny” dodawania liczb binarnych.

* 1. Odejmowanie tablicowe

Kolejna funkcja używana przy mnożeniu to *subtractArrays()*, która wykonuje odejmowanie binarne dwóch tablic przekazanych w parametrach. Jej działanie można porównać do sposobu “pisemnego”, gdzie odejmowane są bity od strony najmniej znaczącej części liczb, a gdy nie można odjąć 1 od 0, następuje pożyczka ze starszej części.

* 1. Mnożenie

Funkcja *multiply()* wykonuje mnożenie dwóch liczb wykorzystując algorytm mnożenia “pisemnego”, gdzie każda cyfra drugiego czynnika, zaczynając od najmłodszej pozycji, mnożona jest po kolei przez każdą cyfrę pierwszego czynnika, również od najmłodszej pozycji. Po jednej takiej operacji otrzymamy jeden rząd, który później należy dodać do pozostałych, powstałych z kolejnych mnożeń, aby uzyskać końcowy wynik. W tym przypadku nie mnożona jest cyfra a cała tablica *unsigned int*.

Argumentami funkcji *multiply()* są wektory przechowujące binarne wartości liczb wczytanych na wejściu do programu.

Na początku, funkcja sprawdza czy któryś z czynników jest równy 0, po czym od razu zwraca wynik 0. Gdy wartości liczb są niezerowe, obliczana jest ilość miejsc (cyfr) po przecinku w obydwu liczbach, na podstawie których odpowiednio dobierana jest ilość miejsc po przecinku dla wyniku.

Mnożenie kontenera jednego czynnika przez pozostałe drugiego czynnika odbywa się za pomocą wstawki asemblerowej.

Po wykonaniu mnożenia, niższa część wyniku znajduje się w *resultNumberArray[0]* natomiast wyższa w *resultNumberArray[1]* (ta część jest już w kolejnej tablicy *unsigned int*). Wynik przekazywany jest do funkcji *sum\_up\_result()*, po której działaniu otrzymujemy cały jeden rząd.

Po wykonaniu wszystkich mnożeń w obu czynnikach, wywoływana jest funkcja *sum\_up\_number\_rows()*, która dodaje do siebie wszystkie rzędy powstałe przy mnożeniu, po czym w wektorze *finalResult* umieszcza końcowy wynik mnożenia.

void sum\_up\_number\_rows(vector<unsigned int> &resultRow, vector<unsigned int> addRow,int counter)

{

unsigned int digit;

for(int i = 0;i<addRow.size();i++)

{

if(resultRow.size() > i+counter)

{

digit = addRow.at(i);

for(int j=counter+i; j<resultRow.size();j++)

{

if(\_\_builtin\_uadd\_overflow(resultRow.at(j), digit, &resultRow.at(j)))

{

digit=1;

if(j+1==resultRow.size())

{

resultRow.push\_back(1);

break;

}

else

continue;

}

else

{

break;

}

}

}

else

resultRow.push\_back(addRow.at(i));

}

}

Listing 4. Kod źródłowy funkcji *sum\_up\_number\_rows()*.

Po uzyskaniu iloczynu, sumowane są eksponenty za pomocą funkcji *addArrays()*.

numberIEEE multiply(numberIEEE n1, numberIEEE n2)

{

unsigned int resultNumberArray[2],l1,l2;

vector <unsigned int> result;

vector <vector<unsigned int>> resultsArray;

vector <unsigned int> finalResult;

numberIEEE number;

if((n1.number.size()==1 && n1.number.at(0)==0) || (n2.number.size()==1 && n2.number.at(0)==0))

{

numberIEEE n('+',"0",'+',"0");

return n;

}

unsigned int numLength = n1.number.size()-1 + n2.number.size()-1;

unsigned int bitsLengthLastContainer=0;

for(int i =31; i>=0;i--)

{

if(readBit(n1.number.at(n1.number.size()-1), i))

{

bitsLengthLastContainer+=i+1;

break;

}

}

for(int i =31; i>=0;i--)

{

if(readBit(n2.number.at(n2.number.size()-1), i))

{

bitsLengthLastContainer+=i+1;

break;

}

}

if(bitsLengthLastContainer / 32 > 0)

{

bitsLengthLastContainer = bitsLengthLastContainer % 32;

numLength++;

}

for(int j = 0; j<n2.number.size();j++)

{

vector <unsigned int> result;

for(int i = 0; i<n1.number.size();i++)

{

l1=n1.number[i]; l2=n2.number[j];

asm ("movl %2, %%eax;"

"mull %3;"

"movl %%eax, %0;"

"movl %%edx, %1;"

: "=r" ( resultNumberArray[0] ), "=r" ( resultNumberArray[1] )

: "r" ( l1 ), "r" ( l2 )

:"eax","edx"

);

sum\_up\_result(i, resultNumberArray, result);

}

if(result.back()==0)

result.pop\_back();

resultsArray.push\_back(result);

for(int i=0;i<resultsArray.size();i++)

sum\_up\_number\_rows(finalResult,resultsArray.at(i), i);

numberIEEE newNumber;

newNumber.number=finalResult;

int bitSizeOfResult=31;

for(; bitSizeOfResult>=0;bitSizeOfResult--)

{

if(readBit( newNumber.number.at( newNumber.number.size()-1), bitSizeOfResult))

{

bitSizeOfResult++;

break;

}

}

if(numLength!=finalResult.size()-1)

cout<<"numLength:"<<numLength<<"ERR(((!=))) bitsLengthLastContainer: "<<bitsLengthLastContainer<<endl;

if(int(n1.numberSign+n2.numberSign)==int('+'+'-'))

newNumber.numberSign='-';

else if(int(n1.numberSign+n2.numberSign)==int('-'+'-'))

newNumber.numberSign='+';

else

newNumber.numberSign='+';

newNumber.exponent=n1.exponent;

if(int(n1.exponentSign+n2.exponentSign)==int('+'+'-'))

{

if(isSubstractable(n1.exponent,n2.exponent))

{

substractArrays(newNumber.exponent, n2.exponent);

if(n1.exponentSign=='+')

newNumber.exponentSign='+';

else

newNumber.exponentSign='-';

}

else

{

newNumber.exponent=n2.exponent;

substractArrays(newNumber.exponent, n1.exponent);

if(n2.exponentSign=='+')

newNumber.exponentSign='+';

else

newNumber.exponentSign='-';

}

}

else if(int(n1.exponentSign+n2.exponentSign)==int('-'+'-'))

{

addArrays(newNumber.exponent,n2.exponent);

newNumber.exponentSign='-';

}

else

{

addArrays(newNumber.exponent,n2.exponent);

newNumber.exponentSign='+';

}

if(bitSizeOfResult>bitsLengthLastContainer)

{

vector<unsigned int> b;

b.push\_back(1);

if(newNumber.numberSign=='+')

addArrays(newNumber.exponent, b);

else

substractArrays(newNumber.exponent, b);

}

if(newNumber.exponent.at(0)==0 && newNumber.exponent.size()==1)

newNumber.exponentSign='+';

return newNumber;

}

Listing 5. Kod źródłowy funkcji *multiply()*.

* 1. Dzielenie

Funkcja *divide()* odpowiada za wykonanie dzielenia wczytanych do programu liczb. Są one przekazane, jako argumenty po przeprowadzeniu wcześniejszej konwersji na system binarny. Pierwszy argument to dzielna, natomiast drugi to dzielnik. Na samym początku, sprawdzana jest wartość dzielnika, po czym wyświetlany stosowny komunikat i wyjście z funkcji, gdy wynosi ona zero.

Następnie, inicjujemy resztę *residue*,jako wartość 0. Algorytm rozpoczyna pracę od najstarszego bitu. Jego działanie oparte jest na dzieleniu nieodtwarzającym liczb binarnych, z wykorzystaniem operacji odejmowania. Pobieramy kolejno bity z dzielnej za pomocą operacji ‘shift’ funkcją *shiftArrayLeft()* i umieszczamy w buforze, tutaj: *residue*. Metoda *bool* *isSubstractable()* sprawdza na każdym kroku pętli, czy wartość znajdująca się w *residue* jest większa, bądź równa wartości dzielnika. W momencie, gdy warunek ten jest spełniony, do bufora *residue* wędrują wartości ‘1’ typu *string*. Po wykonaniu odejmowania funkcją *subtractArrays()*, różnica jest umieszczana właśnie w przekazanym wektorze *residue*. Dalej ponownie wykonywana jest operacja lewostronnego ‘shiftowania’, umieszczania bitów dzielnej w *residue* i sprawdzanie warunku *isSubstractable()*. Trwa to aż do uzyskania wszystkich bitów dzielnej.

numberIEEE divide(numberIEEE n, numberIEEE divisor){

vector<unsigned int> residue;

residue.push\_back(0);

numberIEEE result;

result.number.push\_back(0);

if(divisor.number.size()==1 && divisor.number.at(0)==0)

{

cerr<<"BLAD!: DZIELENIE PRZEZ 0"<<endl;

numberIEEE n('+',"0",'0',"0");

return n;

}

for(int i = n.number.size()-1; i>=0;i--)

{

for(int bit=31;bit>=0; bit--)

{

if(readBit(n.number.at(i),bit))

{

shiftArrayLeft(residue, true);

}

else

{

shiftArrayLeft(residue, false);

}

if(isSubstractable(residue, divisor.number))

{

shiftArrayLeft(result.number,true);

substractArrays(residue, divisor.number);

}

else

{

shiftArrayLeft(result.number,false);

}

}

}

if(int(n.numberSign+divisor.numberSign)==int('+'+'-'))

result.numberSign='-';

else if(int(n.numberSign+divisor.numberSign)==int('-'+'-'))

result.numberSign='+';

else

result.numberSign='+';

result.exponent=n.exponent;

if(int(n.exponentSign+divisor.exponentSign)==int('+'+'-'))

{

addArrays(result.exponent,divisor.exponent);

if(n.exponentSign=='+')

result.exponentSign='+';

else

result.exponentSign='-';

}

else if(int(n.exponentSign+divisor.exponentSign)==int('-'+'-'))

{

if(isSubstractable(n.exponent,divisor.exponent))

{

substractArrays( result.exponent, divisor.exponent);

result.exponentSign='+';

}

else

{

result.exponent=divisor.exponent;

substractArrays(result.exponent, n.exponent);

result.exponentSign='-';

}

}

else

{

if(isSubstractable(n.exponent,divisor.exponent))

{

substractArrays( result.exponent, divisor.exponent);

result.exponentSign='+';

}

else

{

result.exponent=divisor.exponent;

substractArrays(result.exponent, n.exponent);

result.exponentSign='-';

}

}

if(result.exponent.at(0)==0 && result.exponent.size()==1)

result.exponentSign='+';

if(result.number.at(0)==0 && result.number.size()==1)

cerr<<"BLAD! : DZIELENIE POZA ZAKRESEM (MNIEJSZA\_LICZBA/WIEKSZA\_LICZBA)"<<endl;

return result;

}

Listing 6. Kod żródłowy funkcji *divide()*.

Po uzyskaniu ilorazu, odejmowane są eksponenty obu liczb, gdzie również wykorzystywana jest funkcja *subtractArrays()*.

* 1. Wyświetlenie wyniku w postaci szesnastkowej

Po wczytaniu, konwersji na system binarny i umieszczeniu w tablicach *unsigned int*, liczby zostają przygotowane do wyświetlenia w reprezentacji szesnastkowej i zmiennoprzecinkowej. Odpowiada za to funkcja *toStringHex()*, w której wykonywanych jest kilka operacji. Najpierw w zmiennej *int* zapamiętywana jest reszta z dzielenia ostatnich dwóch bitów eksponenty przez 4 (wykonywana jest operacja modulo 4). W celu otrzymania kodu szesnastkowego, wykonywana jest za pomocą funkcji *shiftArrayRight()* dwukrotna operacja ‘shift’ w prawą stronę (czyli dzielenie przez 4), następnie funkcja *shiftArrayLeft()* wykonuje się tyle razy, ile wynosiło powyższe modulo, co oznacza mnożenie mantysy przez 2 do właśnie tej potęgi. Ten zabieg zapewnia prawidłową wartość mantysy przy wyświetlaniu. Obecna tutaj również funkcja *print\_vectorHex()* zwraca podaną liczbę *unsigned int* w postaci szesnastkowej jako *string*. Dalej formowana jest odpowiednia notacja, czyli usuwanie początkowych zer, dodawanie znaku przecinka na drugim miejscu i dopisywany do liczby znak minusa w zależności od tego, czy wprowadzono go na wejściu.

string toStringHEX()

{

string text,text\_buffor;

vector <unsigned int> exponent\_buffor = exponent;

vector <unsigned int> number\_buffor = number;

int end;

int rest = exponent\_buffor.at(0) % 4;

for(int i=0;i<2;i++)

shiftArrayRight(exponent\_buffor);

for(int i=0;i<rest;i++)

shiftArrayLeft(number\_buffor,false);

if(number\_buffor.at(0)==0)

return "0";

if(numberSign=='-')

text+="-";

text\_buffor=print\_vectorHex(number\_buffor);

end = text\_buffor.find\_first\_of("123456789abcdefABCDEF");

text\_buffor.erase(0, end);

text\_buffor.insert(1,",");

if(text\_buffor.size()==2)

text\_buffor.pop\_back();;

text+=text\_buffor;

if(!(exponent\_buffor.size()==1 && exponent\_buffor.at(0)==0))

text+=" \* 10^";

if(exponentSign=='-')

text+="-";

text\_buffor=print\_vectorHex(exponent\_buffor);

end = text\_buffor.find\_first\_of("123456789abcdefABCDEF");

text\_buffor.erase(0, end);

text+=text\_buffor;

return text;

}

Listing 7. Kod źródłowy funkcji *toStringHEX()*.

1. Działanie programu - testy
   1. Mnożenie

Przeprowadzono testy działania operacji mnożenia dla następujących danych wejściowych:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Liczba1** | | **Liczba2** | | **Wynik** |
| **Mantysa** | **Eksponenta** | **Mantysa** | **Eksponenta** |
| +0000.00000 | +0.0000 | +5 | +15 | 0 |
| +5.02 | +0 | +0 | +0 | 0 |
| -20.5 | +5 | +12.34 | -6 | -2,4c304\*10^1 |
| + fffffe,abfefe | +0 | + fefefe,98213b | +0 | 1,fdfdfa8aeb1fc1ee90a50514\*10^a |
| +1.11101011 | +0 | +1.01110110 | +0 | 1,1233435443321210 |
| +fffffffffffffffffff | +0 | +1000000,000000001 | +0 | 1,000000000000000fffefffffffffffffff \* 10^18 |

Tabela 1. Wyniki działań mnożenia na liczbach zmiennoprzecinkowych.

Wyniki testów dla danych większego rozmiaru:

|  |  |
| --- | --- |
| **Mantysa1:** | +FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1 |
| **Eksponenta1:** | +0 |
| **Mantysa2:** | 321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812 |
| **Eksponenta2:** | +0 |
| **Wynik:** | 3,1e4b423049a0200ef42c8f3b77ad9d3aca0ddc46a5bb1a669fef2951d3c8979275d0765d01d614be4bb1c3682fe391ea219310735df10f15f7745d7e8bfe8c41cd55aa89ba0c096da336f794e8198699791844a0162703c54ef991ab443480f124dadeb67241fe1cfabc2bc1a04f7b48d09d78ccce5cf871883383a7b2ca558e69e84177692a35829406f46c3b1cb856be25a7610d0f3b2ae8445a55df01bdff12630d4ab0f440d33c81c03f82e6c3a766a0733454d9467b90bf262926cbc94fbaddd91df8be4c23e4fc8c12cab0cef80f1b3f079ca351cc3939f1fc6e95d4a06358a4f1408857748d7757e6127ada48b7960adae46d5d2 \* 10^1ee |
| **Mantysa1:** | +1e4b423049a0200ef42c8f3b77ad9d3aca0ddc46a5bb1a669fef2951d3c8979275d0765d01d614be4bb1c3682fe391ea219310735df10f15f7745d7e8bfe8c41cd55aa89ba0c096da336f794e8198699791844a0162703c54ef991ab443480f124dadeb67241fe1cfabc2bc1a04f7b48d09d78ccce5cf871883383a7b2ca558e69e84177692a35829406f46c3b1cb856be25a7610d0f3b2ae8445a55df01bdff12630d4ab0f440d33c81c03f82e6c3a766a0733454d9467b90bf262926cbc94fbaddd91df8be4c23e4fc8c12cab0cef80f1b3f079ca351cc3939f1fc6e95d4a06358a4f1408857748d7757e6127ada48b7960adae46d5d2 **,** eeffebfcd1627378fe |
| **Eksponenta1:** | +0 |
| **Mantysa2:** | dc46a5bb1a669fef2951d3c8979275d0765d01d614be4bb1c3680200ef42c8f3b77ad9d3aca0ddc44b423049a0200ef42c8f3b77ad9d3aca0ddc46 **,** 213321dcda213 |
| **Eksponenta2:** | +0 |
| **Wynik:** | 1, \* 10^263 |

Tabela 2. Wyniki działań mnożenia na liczbach dużego rozmiaru.

* 1. Dzielenie

Przeprowadzono testy działania operacji dzielenia dla następujących danych wejściowych:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Liczba1** | | **Liczba2** | | **Wynik** |
| **Mantysa** | **Eksponenta** | **Mantysa** | **Eksponenta** |
| +0000.00000 | +0.0000 | +5 | +15 | 0, błąd: dzielenie mniejsza/większa |
| +5.02 | +0 | +0 | +0 | Błąd: dzielenie przez 0 |
| -20.5 | +5 | +12.34 | -6 | -9\*10^-b |
| + fffffe,abfefe | +0 | + fefefe,98213b | +0 | 1 |
| +1.11101011 | +0 | +1.01110110 | +0 | 1 |
| +fffffffffffffffffff | +0 | +1000000,000000001 | +0 | f,fff \* 10^c |

Tabela 3. Wyniki dzielenia na liczbach zmiennoprzecinkowych.

Wyniki testów dla danych większego rozmiaru:

|  |  |
| --- | --- |
| **Mantysa1:** | +FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1FEFDCA986FE67FE1 |
| **Eksponenta1:** | +0 |
| **Mantysa2:** | 321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812321739FE89ACD812 |
| **Eksponenta2:** | +0 |
| **Wynik:** | 5,1730eb6499eee427 \* 10^10 |
| **Mantysa1:** | +1e4b423049a0200ef42c8f3b77ad9d3aca0ddc46a5bb1a669fef2951d3c8979275d0765d01d614be4bb1c3682fe391ea219310735df10f15f7745d7e8bfe8c41cd55aa89ba0c096da336f794e8198699791844a0162703c54ef991ab443480f124dadeb67241fe1cfabc2bc1a04f7b48d09d78ccce5cf871883383a7b2ca558e69e84177692a35829406f46c3b1cb856be25a7610d0f3b2ae8445a55df01bdff12630d4ab0f440d33c81c03f82e6c3a766a0733454d9467b90bf262926cbc94fbaddd91df8be4c23e4fc8c12cab0cef80f1b3f079ca351cc3939f1fc6e95d4a06358a4f1408857748d7757e6127ada48b7960adae46d5d2 **,** eeffebfcd1627378fe |
| **Eksponenta1:** | +0 |
| **Mantysa2:** | dc46a5bb1a669fef2951d3c8979275d0765d01d614be4bb1c3680200ef42c8f3b77ad9d3aca0ddc44b423049a0200ef42c8f3b77ad9d3aca0ddc46 **,** 213321dcda213 |
| **Eksponenta2:** | +0 |
| **Wynik:** | 2,334feb9e33ae61602b468968379848d52361bbdd222ae3f5d4acdc4110d5c23d250bca8aa53aaf693d29316a9b68d5862dc59626003bb005e116b713e6ab0ab1b93e67f279dffd448bbcde55b48ba0cbc8825ce2c2cbf7bf60fda2211cd0779439da140379d27269fa0277f89b7b1575d34ea3e8c6fbb050d3b0648b8d00d98cecf4cec7126d8f937c68d23437fa70c1a04170162793b79fae61ebedd63334ae14043820a58984f2735d722844d842f819fe0e5490fe6f4e3b0fc2949f5f2 \* 10^179 |

Tabela 4. Wyniki dzielenia dla argumentów dużego rozmiaru.

* 1. Sprawdzanie poprawności

Poprawność wyników działań przeprowadzonych podczas testów zweryfikowano za pomocą kalkulatora dużych liczb, dostępnego na stronie internetowej pod adresem [1].

* 1. Pomiary czasu

Działanie algorytmów mnożenia i dzielenia poddano pomiarom czasowym. Wykorzystano w tym celu instrukcję **RDTSC** (Read Time-Stamp Counter), która ładuje aktualną liczbę cykli procesora, z przechowywującego je rejestru 64-bitowego MSR (Machine State Register), do rejestrów EDX (wyższe 32 bity) i EAX (niższe 32 bity). Wartość zwracana przez funkcję *rdtsc()* ładowana jest to zmiennej, wykonywany jest algorytm mnożenia lub dzielenia, po czym kolejna zmienna przyjmuje bieżącą wartość zwracaną przez *rdtsc()*. Różnica wartości zmiennej po wykonaniu mnożenia/dzielenia i przed wykonaniem zawiera ilość cykli procesora, które wykonał wykonując algorytm. Wartość ta następnie dzielona jest przez taktowanie zegara procesora (tutaj 2494.3, ponieważ podczas pomiarów korzystano z procesora Intel Pentium i3 2,5GHz), a następnie przez 1000 aby otrzymać wynik w milisekundach.

uint64\_t rdtsc(){

unsigned int lo,hi;

\_\_asm\_\_ \_\_volatile\_\_ ("rdtsc" : "=a" (lo), "=d" (hi));

return ((uint64\_t)hi << 32) | lo;

}

Listing 8. Konstrukcja funkcji *rdtsc()*.

Wyniki pomiarów czasowych:

|  |  |  |
| --- | --- | --- |
| **Algorytm mnożenia** | | |
| **Rozmiar**  **argumentów:** | | **Czas wykonywania algorytmu [ms]:** |
| **Czynnik** | **Czynnik** |
| 128 bit | 128 bit | 0,681179 |
| 256 bit | 256 bit | 1,35264 |
| 512 bit | 512 bit | 3,76255 |
| 512 bit | 256 bit | 2,32762 |
| 1024 bit | 1024bit | 10,7352 |
| 1024 bit | 512 bit | 4,90598 |
| 1024 bit | 256 bit | 2,2929 |
| 1536 bit | 1536 bit | 25,8815 |
| 1536 bit | 1024 bit | 14,8414 |
| 1536 bit | 512 bit | 5,89734 |
| 2048 bit | 2048 bit | 34,8024 |
| 2048 bit | 1536 bit | 22,3465 |
| 2048 bit | 1024 bit | 17,5496 |
| 2048 bit | 512 bit | 8,53334 |
| 8192 bit | 512 bit | 59,8063 |

Tabela 5. Wyniki pomiarów czasowych dla algorytmu mnożenia dla różnych rozmiarów argumentów.

|  |  |  |
| --- | --- | --- |
| **Algorytm dzielenia** | | |
| **Rozmiar**  **argumentów:** | | **Czas wykonywania algorytmu [ms]:** |
| **Dzielna** | **Dzielnik** |
| 128 bit | 128 bit | 1,62655 |
| 256 bit | 256 bit | 3,95311 |
| 512 bit | 512 bit | 20,2963 |
| 512 bit | 256 bit | 18,3306 |
| 1024 bit | 1024bit | 30,1164 |
| 1024 bit | 512 bit | 27,7851 |
| 1024 bit | 256 bit | 23,7462 |
| 1536 bit | 1536 bit | 62,6098 |
| 1536 bit | 1024 bit | 60,1759 |
| 1536 bit | 512 bit | 57,0838 |
| 2048 bit | 2048 bit | 93,0740 |
| 2048 bit | 1536 bit | 89,5398 |
| 2048 bit | 1024 bit | 88,9352 |
| 2048 bit | 512 bit | 83,9632 |
| 8192 bit | 512 bit | 1145,48 |

Tabela 6. Wyniki pomiarów czasowych dla algorytmu dzielenia dla różnych rozmiarów argumentów

Powyższe pomiary porównano z algorytmami dzielenia i mnożenia zaimplementowanymi w bibliotece BigInt języka JAVA. Wykorzystano funkcje *multiply* oraz *divide*. Wyniki pomiarów:

|  |  |  |  |
| --- | --- | --- | --- |
| **BigInt** | | **Nasz algorytm** | |
| **Mnożenie** | **Dzielenie** | **Mnożenie** | **Dzielenie** |
| **2048 bit** | | | |
| 3.682ms | 173.319ms | 34,807ms | 93,074ms |
| **1024 bit** | | | |
| 1.327ms | 65.995ms | 10,735ms | 30,116ms |
| **512 bit** | | | |
| 0.396ms | 25.140ms | 3,762ms | 20,296ms |

Tabela 7. Porównanie czasów wykonywania algorytmów mnożenia i dzielenia z biblioteką BigInt języka Java.

1. Wnioski

Największą trudnością w napisaniu programu tego typu jest zrozumienie zapisu liczby binarnej zamieszczonej w wielu tablicach. Poszczególne tablice w systemie binarnym nie odpowiadają reprezentatywnej części tej liczby, a są jej realną częścią. Rozwiązanie, które polegałoby na przedstawianiu tablic reprezentatywnie (zakładając dziesiętnie) byłoby bardzo nieefektywne, słowa poszczególnych tablic musiałby być ograniczone do liczby 999 999 999(10) co pozawala na wykorzystanie jedynie ~23% pojemności całego słowa 32 bitowego.  
Aby efektywnie rozwiązać ten problem, należało wykorzystać fakt, iż komputery przetwarzają dane w systemie binarnym, a podstawowe operacje arytmetyczne zawsze zajmują taką samą liczbę cykli.

Własna interpretacja systemu zmiennoprzecinkowego nie sprawiła dużych problemów. Niestety przez opóźnienie związane z problemem zaimplementowania dzielenia, ta operacja działa tylko na liczbach całkowitych, przez co wraz ze zmniejszeniem stosunku dzielnej do dzielnika, zmniejsza się precyzja wyniku dzielenia. Rozwiązanie tego problemu jest dość proste, jednakże zabrakło czasu na zaimplementowanie go: należało przedłużyć wektor z dzielną o tablicę *unsigned int* (wielkości np. dzielnika) zawierającą same zera, przedłużając w ten sposób dzielenie liczby. Należałoby jednak pamiętać o przecinku, który należy uwzględnić w eksponencie w momencie przejścia dzielenia na taką dodatkową tablice zer, a także odpowiednio po obliczeniach, znormalizować liczbę i przesunąć odpowiednio przecinek za pierwsze wystąpienie ‘1’.

Czas wykonywania algorytmów rośnie proporcjonalnie do ilości bitów argumentów. Algorytm dzielenia jest wolniejszy gdyż wykonuje on działania operując bit po bicie, natomiast algorytm mnożenia wykonuje działania na całych tablicach 32-bitowych. Podczas dzielenia, na czas wykonywania wpływa głównie dzielna.

1. Literatura

[1] Online Big Number Calculator, https://defuse.ca/big-number-calculator.htm

[2] Algorytm Hornera, http://eff10.internetdsl.tpnet.pl/programowanie/kurs\_sc/liczby/pages/03.htm

[3] Built-in Functions to Perform Arithmetic with Overflow Checking, https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html

[4] RDTSC: https://msdn.microsoft.com/pl-pl/library/twchhe95.aspx, http://x86.renejeschke.de/html/file\_module\_x86\_id\_278.html