Organizacja i Architektura Komputerów Projekt

Dokumentacja końcowa

Michał Chojnacki 225936
Tomasz Osada 242085

Prowadzący:
dr inż. Tadeusz Tomczak
Temat: Implementacja biblioteki arytmetyki stałoprzecinkowej dowolnej precyzji z wykorzystaniem wewnętrznej reprezentacji znak-moduł (NB + znak)

23 maja 2019

Spis treści

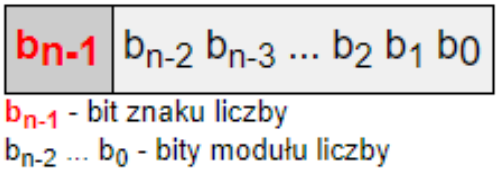
Lista kodów źródłowych

1. Cel projektu

Celem niniejszego projektu jest implementacja biblioteki w języku C++, która umożliwia wykonywanie operacji arytmetycznych, dla liczb o reprezentacji w systemie znak-moduł (ang. Sign-Magnitude).

2. System znak-moduł

System znak-moduł, to jeden z najbardziej intuicyjnych sposóbów zapisu liczby. Osobny zapis znaku i wartości bezwzględnej pozwala na duże uproszczenia, w porównywaniu rzeczywistej odległości od 0 liczb, zapisanych w tej reprezentacji. Nie jest to jednak system, powszechnie stosowany w maszynach, wykonujących operacje arytmetyczne. W modelu IA-32, dla liczb ze znakiem, stosuje się reprezentację uzupełnieniową (opisana w punkcie 4.2.1.2 Signed Integers) i zmiennoprzecinkową (4.2.2). Użytkownicy, implementujący system znak-moduł, najczęściej wykorzystują poniższą interpretację. Liczba kodowana jest w postaci ciagu bitów, gdzie bit o najwyższej wadze jest bitem znaku, pozostałe bity przechowują wartość modułu w systemie naturalnym binarnym.



Rysunek 1: Zapis Z-M

2.1. Zalety

- Zbliżony do systemu zapisu liczb, używanego na co dzień
- Symetryczny zakres

2.2. Wady

- Utrudnione wykonywanie działań arytmetycznych
- Podwójna reprezentacja 0 (+0 i -0)

2.3. Wykonywanie działań

Reguły dodawania liczb ZM wynik = $a_{(ZM)} + b_{(ZM)}$			
Znak $a_{(ZM)}$	Znak $b_{(ZM)}$	operacja	Znak wyniku
0	0	dodawanie modułów	0
1	1	dodawanie modułów	1
0	1	odejmowanie modułu mniejszego od modułu większego	znak większego modułu
1	0		

Rysunek 2: Dodawanie w Z-M

Reguły odejmowania liczb ZM wynik = $a_{(ZM)} - b_{(ZM)}$			
Znak $a_{(ZM)}$	Znak $b_{(ZM)}$	operacja	Znak wyniku
0	0	odejmowanie modułu mniejszego od modułu większego	Znak $a_{(ZM)}$, jeśli moduł ten jest większy od modułu $b_{(ZM)}$. Inaczej znak przeciwny.
1	1		
0	1	dodawanie modułów	0
1	0	dodawanie modułów	1

Rysunek 3: Odejmowanie w Z-M

Reguły mnożenia i dzielenia liczb Z-M wynik = $a_{(ZM)} \times b_{(ZM)}$ wynik = $a_{(ZM)} : b_{(ZM)}$			
Znak $a_{(ZM)}$	Znak $b_{(ZM)}$	operacja	Znak wyniku
0	0	mnożenie lub dzielenie modułów	Jeśli znaki $a_{(ZM)}$ i $b_{(ZM)}$ są takie same, to 0. Inaczej 1
1	1		
0	1		
1	0		

Rysunek 4: Mnożenie i dzielenie w Z-M

3. Założenia projektowe

- Implementacja w C++ (nowoczesne standardy >cpp11)
- znak jako zmienna typu bool (false/0 oznacza +, true/1 oznacza -)
- moduł jako kontener vector, który przechowuje elementy typu unsigned integer, o długości 32 bitów.
- vector zapewnia dowolną rozszerzalność struktury, ograniczoną jedynie przez pamięć maszyny
- Podstawowe operacje arytmetyczne - dodawanie, odejmowanie, mnożenie i dzielenie
- Optymalizacje niskopoziomowe - użycie arytmetycznych instrukcji procesora, dla systemu binarnego (opisane w 5.1.2, Binary Arithmetic Instructions, szerzej opisane w II tomie dokumentacji) oraz zastosowanie flag optymalizacyjnych kompilatora gcc.

3.1. Struktury danych

System znak-moduł rozszerza funkcjonalności systemu naturalnego, dlatego w celu uproszczenia projektu i zapewnienia możliwości rozwoju w innych kierunkach, jako krok pośredni, zaimplementowano reprezentację liczb naturalnych. Została ona przygotowana w formie klasy, która zawiera pole typu vector, mający przechowywać informacje o wartości liczby.

Kod źródłowy 1: Zapis liczby naturalnej

```
class natural {
private:
        std::vector<uint32_t> value;
        ...
};
```

W ten sposób, liczbę naturalną wystarczy wzbogacić o pole znaku i rozszerzyć jej metody, w oparciu o instrukcje warunkowe, związane ze znakami argumentów. Do zapisu znaku, użyto zmiennej typu bool. Zajmuje ona 1 bajt w pamięci komputera (w praktyce korzysta tylko z 1 bitu, jednak nie jest możliwe z poziomu programu zająć mniej niż bajt).

Kod źródłowy 2: Zapis liczby ze znakiem

```
class smNum
{
private:
        bool sign; //false +, true -
        natural module; //modul reprezentowany przez liczbe naturalna
};
```

4. Podstawowe operacje arytmetyczne

By realizować dwuargumentowe dodawanie liczb S-M, należy wykonywać odpowiednie operacje na modułach, przy sprawdzeniu odpowiednich warunków.

Kod źródłowy 3: Operacje arytmetyczne dla systemu Znak-Moduł

```
// dodawanie
void smNum::add(smNum x, smNum y) {
        if (x.sign == y.sign) { //znaki zgodne
                this->sign = x.sign;
                this->module.add(x.module, y.module);
        }
        else { //znaki niezgodne
                if (x.module >= y.module) {
                        this->sign = x.sign;
                        this->module.subtract(x.module, y.module);
                }
                else
                {
                        this->sign = y.sign;
                        this->module.subtract(y.module, x.module);
                }
        }
}
// odejmowanie
void smNum::sub(smNum x, smNum y) {
        if (x.sign == y.sign) { //znaki zgodne
                if (x.module >= y.module) {
                        this->sign = x.sign;
                        this->module.subtract(x.module, y.module);
                }
```

```
                else
                {
                        this->sign = !(x.sign);
                        this->module.subtract(y.module, x.module);
                }
        }
        else { //znaki niezgodne
                this->sign = x.sign;
                if (x.sign == 0) {
                        this->module.add(x.module, y.module);
                }
                else
                {
                        this->module.add(x.module, y.module);
                }
        }
}
// mnozenie
void smNum::mul(smNum x, smNum y) {
                if (x.sign == y.sign)
                        this->sign = 0;
                else
                        this->sign = 1;

                this->module.multiply(x.module, y.module);
}
// prymitywny algorytm dzielenia
void smNum::div(smNum x, smNum y) {
        if (x.sign == y.sign)
                this->sign = 0;
        else
                this->sign = 1;

        this->module.divide(x.module, y.module);
}
```

4.1. Implementacja dodawania i odejmowania dla systemu naturalnego

Funkcje add i subtract zostały napisane według najprostszych zasad arytmetyki.

- Poszczególne 32-bitowe elementy są dodawane sekwencyjnie na odpowiadających sobie pozycjach
- W przypadku, kiedy wyniku nie można zapisać na danej pozycji, generuje się przeniesienie/pożyczkę.
- Dodawanie stosuje własność przemienności, by usprawnić proces przepisywania dłuższej liczby, w przypadku wyjścia poza zakres krótszej.
- Weryfikacja wyniku odejmowania polega na sprawdzeniu czy za pozycją o najwyższej wadze odjemnej występuje pożyczka.

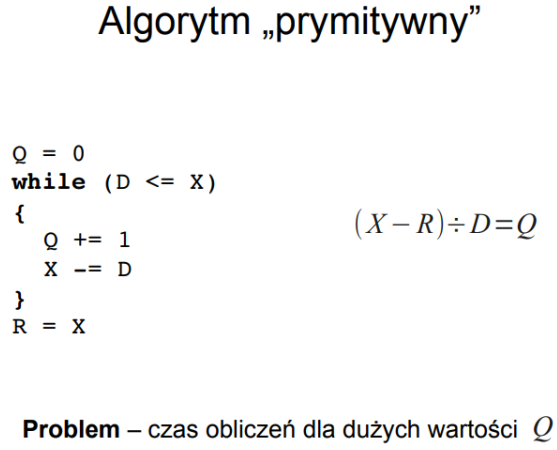
4.2. Implementacja mnożenia dla systemu naturalnego

Funkcja multiply sprawiła więcej problemów projektowych, niż dotychczasowe procedury. Działa ona według następującego schematu

- Wyznacza iloczyny częściowe,iterując po wektorach (argumentach) i dodaje do nich przeniesienie z poprzedniej pozycji.
- Sprawdzenie czy wystąpiło przeniesienie polega na rzutowaniu unsigned long long (64-bity) na usigned int (32-bity)

4.3. Implementacja dzielenia dla systemu naturalnego

Dzielenie jest operacją arytmetyczną, która w największym stopniu, przyczyniła się do zwolnienia tempa pracy nad projektem. W związku z problemami w implementacji korzystniejszych czasowo i pamięciowo algorytmów zdecydowano by umieścić w projekcie algorytm prymitywny, którego prosty schemat jest widoczny poniżej.



Rysunek 5: Opis algorytmu dzielenia, Arytmetyka komputerów

- Zapętlone odejmowanie dzielnika wykorzystuje wcześniej napisaną funkcję odejmowania dla liczb typu natural. Operacja porównania korzysta z przeciążonych operatorów ”>” oraz ”==” (pojedynczy operator >= wykorzystuje więcej porównań niż >, dlatego >= został w tym przypadku rozbity)
- Na rzecz poprawności wykonywania działań, napisano dodatkowe funkcje pomocnicze - isZero(), która zapobiega dzieleniu przez reprezentację 0, oraz eraseLeadingZeroIfExists(), która zapobiega ”fałszowaniu” wyników porównań.

5. Środowisko i uruchomienie

Biblioteka została napisana z myślą o systemach operacyjnych, opartych o jądro Linuxa. Nie ma jednak przeciwskazań, by pliki źródłowe zostały zbudowane pod systemem Windows (chociaż może to mieć znaczący wpływ na działanie programu). Do kompilacji użyto GNU Compiler Collection, domyślnego kompilatora dla systemów typu Linux. Dzięki małej liczbie plików źródłowych, na rzecz projektu udało się uniknąć konieczności pisania obszernego pliku Makefile.

Kod źródłowy 4: Makefile dla projektu Sign-Module

```
all:
	gcc main.cpp natural.cpp sm.cpp -Wall -lgtest -lgtest_main \\
	-lstdc++ -O3 -pg -o sign-module
```

main.cpp, natural.cpp i sm.cpp to pliki źródłowe projektu, -Wall jest flagą pomocznicą, która wyświetla wszystkie ostrzeżenia, -lgtest i -lgtest_main umożliwiają zastosowanie biblioteki Google test. -lstdc++ umożliwia kompilację programów, napisanych w języku C++. -O3 zapewnia najwyższy stabilny poziom optymalizacji kodu, -pg generuje informacje potrzebne profilerowi gprof, a -o zwyczajnie zapisuje wynik kompilacji do pliku wykonywalnego. Opcjonalne flagi to -g dla debuggera gdb, -S generowanie źródeł assemblera.

6. Testy

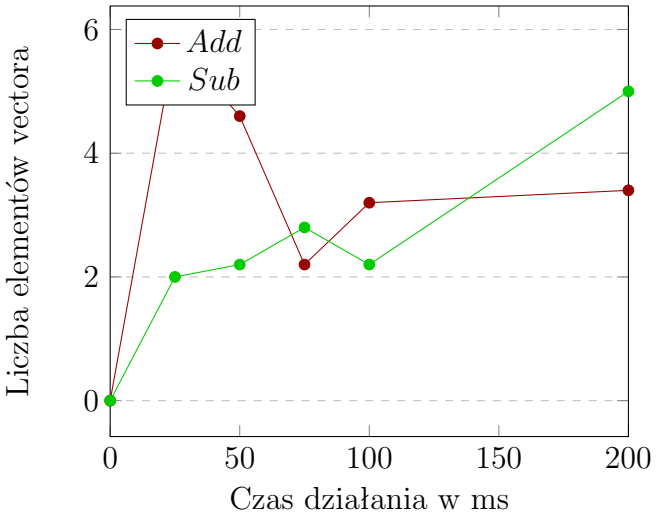
6.1. Testy jednostkowe

Aby upewnić się, że zaimplementowane funkcje działają w sposób poprawny, należało napisać testy jednostkowe. W tym celu zastosowano open-source'owe rozwiązanie firmy Google, dedykowane pod C++, Google Test.

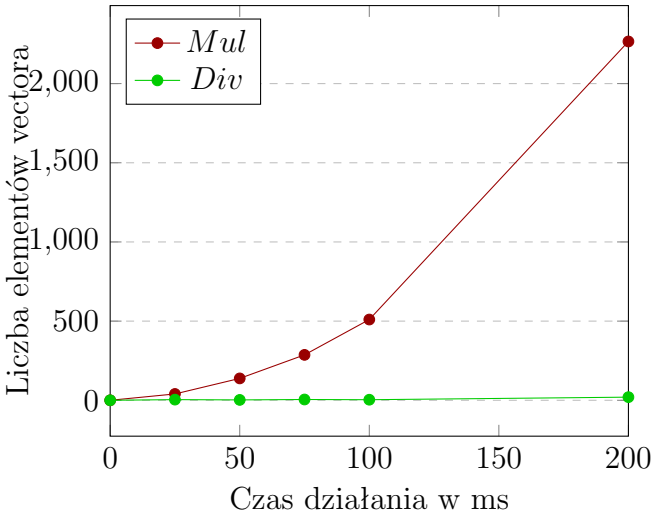
6.2. Profiler

Użycie profilera, dostarczanego przez gcc, odbywa się przy użyciu komendy gprof -p -b ./sign-module lub gprof sign-module gmon.out. W zależności od parametrów wykonania programu, profiler wskazuje operację mnożenia (natural::multiply(const natural &x, const natural &y)) jako najbardziej czasochłonną operację.

6.3. Uśrednione wyniki działania funkcji



Rysunek 6: Czas działania add i sub dla różnych danych wejściowych



Rysunek 7: Czas działania mul i div dla różnych danych wejściowych

7. Wnioski

Przedstawione powyżej wyniki pokazują, że chociaż operacje dodawania i odejmowania są operacjami liniowymi, jakość danych na wejściu może mieć znaczący wpływ przebieg wykonywanych działań, co może dowodzić, że optymalizacje niskopoziomowe nie funkcjonują w sposób prawidłowy dla operacji propagowania przeniesień i pożyczek. Operacja mnożenia, z założenia ma wysoką złożoność obliczeniową, którą dałoby się zrównoleglić i tym samym przyśpieszyć, stosując mechanizmy wielowątkowości. W wyniku obserwacji udało się zauważyć że, operacja dzielenia metodą "prymitywną", jest zależna w bardzo małym stopniu od rozmiaru danych wejściowych. Kluczową rolę w tym algorytmie gra RÓŻNICA w rozmiarze danych wejściowych, która warunkuje liczbę wykonanych operacji odejmowania.

Literatura

[1] Zapis w systmie znak-moduł *https://eduinf.waw.pl/inf/alg/006_bin/0016.php*

[2] Wykłady z Arytmetyki Komputerów *http://zak.ict.pwr.wroc.pl/materials/Arytmetyka\%20komputerow/*

[3] Dokumentacja klasy Big Integer w języku Java *https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html*

[4] Repozytorium Git projektu Google Test *https://github.com/google/googletest*

[5] Cpp reference *https://en.cppreference.com/w/* Dokumentacja kompilatora GCC *https://gcc.gnu.org/onlinedocs/gcc/*

[6] Dokumentacja procesora Intel IA32 *https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf?fbclid=IwAR3Yb7GQcYCJlkBvuIaTBSPfruw2qF6W3IVkBz6Qssy4nANACMZX-UCokiE*

[7] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.