

Organizacja i architektura komputerów

Implementacja procedur obliczeń na liczbach zmiennoprzecinkowych za pomocą instrukcji stałoprzecinkowych.

Mateusz Gurski, 242089
Damian Koper, 241292

Cel projektu

Celem projektu była implementacja procedur obliczeń na liczbach zmiennoprzecinkowych połowicznej i pojedynczej precyzji za pomocą instrukcji stałoprzecinkowych.

Ograniczenia

Jedynym ograniczeniem była długość słowa, ustalona na 8 bitów. Zadanie można interpretować, jako stworzenie biblioteki do programowej realizacji obliczeń zmiennoprzecinkowych, gdzie ograniczenie słowa do 8 bitów sygnalizuje, że może być ona użyta w mikrokontrolerach 8 bitowych, które naturalnie nie mają jednostki zmiennoprzecinkowej, a gdzie zachodzi potrzeba takich obliczeń.

IEEE-754 - Single

Liczba w formacie pojedynczej precyzji przechowywana jest w pamięci w następującym formacie:

znak	ułamek
x xxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxx	
wykładnik	

Ułamek zapisany jest z dodanym obciążeniem, co pozwala zachować ciągłość reprezentacji i ułatwia porównania. Dla formatu single obciążenie to wynosi +127. Jeśli wykładnik nie reprezentuje liczb zdenormalizowanych (wartość 0x00), ułamek zawiera ukrytą jedynkę z przodu.

IEEE-754 - Half

Liczba w formacie połowicznej precyzji przechowywana jest w pamięci w następującym formacie:

znak	ułamek	
x	xxxxx	xxxxxxxxxxxx
	wykładnik	

Obciążenie dla formatu half wynosi +15. Jeśli wykładnik nie reprezentuje liczb zdenormalizowanych (wartość 0b00000), ułamek zawiera ukrytą jedynkę z przodu.

Procedury obliczeń

Dodawanie, odejmowanie

Dodawanie i odejmowanie mogą być zaimplementowane jako jedna operacja, co wynika z zależności $A - B = A + (-B)$. Jako, że implementowane dodawanie jest działaniem przemienne, warto rozpatrywać zawsze jeden przypadek, gdzie $A \leq B$. W przypadku kiedy $A > B$ należy zamienić oba składniki miejscami.

Następnie trzeba wyrównać wykładnik mniejszej liczby. Wiemy, że $A \leq B$, więc trzeba przesunąć bity mantysy B o $\text{exp}[A] - \text{exp}[B]$ w lewo. Na tym etapie, znając utracone bity, możemy zaokrąglić otrzymaną liczbę.

Jeśli znaki obu składników są takie same, należy na mantysach wykonać operację dodawania z zachowaniem znaku wyniku, a jeśli różne, odejmowania razem z ustawieniem znaku wyniku na minus.

Po uzyskaniu wyniku trzeba go znormalizować przesuwając go w lewo lub w prawo jednocześnie zmniejszając lub zwiększając wykładnik wyniku, który początkowo ma wartość $\text{exp}[A]$. Przy wynikach, dla których wartość nie mieści się w przedziale wartości liczb pojedynczej precyzji należy pamiętać o ustawieniu w odpowiednich przypadkach wartości 0 i *inf*. Wartość NaN nie występuje jako wynik w przypadku tych operacji przy poprawnych składnikach.

Mnożenie

Mnożąc liczby ustawiamy znak wyniku według zależności $\text{sign}[C] = \text{sign}[A] \text{ XOR } \text{sign}[B]$. Następnie sprawdzamy, czy którykolwiek ze składników ma wartość 0. Jeśli tak, to zwracamy 0 z odpowiednim znakiem (jeśli chcemy mieć znakowane 0).

Wynikowy wykładnik otrzymamy poprzez dodanie wartości wykładników składników, pamiętając o odjęciu obciążenia, a następnie korygując go w procesie normalizacji. W celu normalizacji iloczynu mantys, wiedząc, że iloczyn liczb 24 bitowych zawsze da wynik maksymalnie 48 bitowy, możemy sprawdzić bit 48 wyniku tego działania. Jeśli ma on wartość 1 oznacza to, że wartość jest za duża. Trzeba zwiększyć wykładnik i przesunąć mantysę w prawo.

W przypadku, gdy wartość jest zbyt mała, trzeba sprawdzać bit 47 iloczynu, zmniejszać wykładnik i przesunąć mantysę w lewo, aż owy bit nie będzie miał wartości 1 lub wykładnik nie będzie równy 0x01 (dla Half 0b000001) - w takim wypadku otrzymamy wynik zdenormalizowany. W przypadku wyniku zdenormalizowanego wykładnik reprezentowany jest jako 0x00 (dla Half 0b000000).

Zaraz po wykonaniu mnożenia mantys, znając pozostałe bity wyniku, które zostaną utracone, możemy wykonać zaokrąglanie. Tak jak w dodawaniu/odejmowaniu, w przypadku przekroczenia zakresu trzeba pamiętać o ustawieniu wartości $\pm inf$.

Dzielenie

Dzielenie odbywa się na podobnej zasadzie co mnożenie.

W tym przypadku ważna jest początkowa walidacja liczb. Rozróżniamy przypadki, którym trzeba ustawić specjalne wartości:

- $A/0 = inf$
- $0/0 = NaN$

Aby uzyskać wykładnik trzeba odjąć od siebie wykładniki dzielnej i dzielnika, a następnie dodać obciążenie. Interpretując mantysę jako liczbę w formacie $Q23$ (dla Half $Q10$), stosując arytmetykę stałoprzecinkową, przesuwając bity dzielnej o 26 (dla Half o 13) w lewo, jako wynik dzielenia $frac[A]/frac[B]$ otrzymamy liczbę w formacie $Q26$ (dla Half $Q13$), co daje nam wymagane 23 (dla Half $Q10$) bity mantysy i trzy dodatkowe bity GRS. Przy czym bit S:

$$S = S \mid \text{mod}(frac[A]/frac[B]) \neq 0$$

uwzględnia bity reszty.

Normalizacja wyniku przebiega podobnie jak w przypadku mnożenia. Różni się tylko rozmiarem wyniku, a co za tym idzie pozycjami bitów, które świadczą o potrzebie normalizacji.

W przypadku przekroczenia zakresu trzeba pamiętać o ustawieniu wartości ± 0 .

Pierwiastek

Pierwiastkowanie możliwe jest tylko, gdy liczba jest dodatnia, więc obliczenia trzeba rozpocząć od sprawdzenia znaku. Następnie trzeba sprawdzić czy wykładnik jest parzysty – jeśli nie, zmniejszamy go o 1 odpowiednio skalując mantysę. Gdy wykładnik jest już parzysty, dzielimy go przez 2 i otrzymujemy w ten sposób wykładnik wyniku. Dzielenie wykładnika przez 2 można łatwo wykonać przesuwając wykładnik, od którego wcześniej odjęto obciążenie, o 1 w prawo.

Kolejnym krokiem jest spierwiastkowanie mantysy. Wykonujemy algorytm

$$(2 * Q_i * B + x) * x \leq R_i$$

gdzie Q_i – aktualny wynik, B – podstawa liczby (w tym przypadku 2), R_i – aktualna reszta, x - kolejny bit wyniku pierwiastkowania.

W przypadku pierwiastkowania liczb o podstawie 2, x równe jest 1 jeśli dla x równego 1 spełnione jest powyższe równanie, jeśli nie – x równe jest 0. Jeśli wykładnik był parzysty to mantysa razem z niejawną jedynką jest teraz postaci $1, x . . x$, jeśli był nieparzysty to mantysa jest postaci $1x, x . . x$, pierwszy krok

algorytmu wykonywany jest więc poza pętlą dla **01** gdy wykładnik był parzysty lub dla **1x** gdy był nieparzysty. Pozostałe kroki algorytmu wykonywane są już w pętli.

Implementacja

Procedury obliczeń realizuje stworzona biblioteka języka **C++**. Architektura projektu została zrealizowana w trzech częściach, gdzie każda korzysta z kolejnej:

```
Simple --> Single/Half --> Single/Half(C++)
```

Simple

Część *Simple* realizuje obliczenia na liczbach stałoprzecinkowych odpowiedniej długości. Dla każdej z nich stworzone zostały funkcje operujące na liczbach różnej długości w celu optymalnego wykorzystania w zależności od potrzeb:

- Add 16/24/32/48b
- Sub 16/24/32/48b
- Div 16/32/64b
- Mul 16/32b
- ShiftL 16/24/32/48b
- ShiftR 16/24/32/48b
- HalfToFloat
- FloatToHalf

Do każdej z nich dostarczane są wskaźniki na dane, poprzez które również zwracany jest wynik. HalfToFloat i FloatToHalf do konwersji wykorzystują rozszerzenie *F16C(SSE)* procesora i instrukcje **VCVTPS2PH** oraz **VCVTPH2PS**. Według ustaleń konwersja nie podlega ograniczeniu długości słowa 8 bit. Korzystanie z tych instrukcji w wyższych warstwach eliminuje całkowicie nałożone ograniczenie długości słowa. Używając one instrukcji operujących na liczbach 8 bitowych, wykorzystujących flagę przeniesienia oraz algorytmy m.in. dzielenia nieodtworzącego.

Single/Half

Część *Single/Half* implementuje obliczenia na liczbach zmiennoprzecinkowych połowicznej i pojedynczej precyzji używając operacji *Simple*. Według opisanych wyżej procedur obliczeń, na poziomie asemblera zaimplementowane zostały operacje dodawania/odejmowania, mnożenia, dzielenia i pierwiastka. Do każdej z funkcji dostarczane są wskaźniki na dane, poprzez które również zwracany jest wynik.

Single/Half (C++)

Produktem końcowym jest biblioteka napisana w języku **C++**. Stworzone zostały dwa typy **floating::Single** i **floating::Half** posiadające takie same API, a różniące się tylko rozmiarem pola danych, które stworzono za pomocą unii w celu uzyskania dostępu do danych w różnych interpretacjach.

```

namespace floating
{
class Single
{
public:
    Single(){};
    Single(float f);
    Single(long double longDouble);
    Single(unsigned long long uLongLong);
    Single(const Single &s);

    operator int();
    operator float();
    operator double();

    int toInt();
    float toFloat();
    double toDouble();

    Single operator-();
    Single operator+(const Single &s);
    Single operator-(const Single &s);
    Single operator*(const Single &s);
    Single operator/(const Single &s);

    Single abs();
    Single changeSign();

    Single add(Single component);
    Single multiply(Single multiplier);
    Single subtract(Single subtrahend);
    Single divideBy(Single divisor);
    Single sqrt();
    bool operator==(const Single &s);
    bool operator==(const float &f);

    std::string printBinary();
    std::string printExponent(bool binary = false);
    std::string printFraction(bool binary = false);

private:
    /**
     *          sign          fraction
     * Format: d | ddddddc | ccccccbbbbbaaaaaaa
     *          exponent
     * Float in memory looks like:
     * [d      c      b      a      ]
     * [bytes[3]bytes[2]bytes[1]bytes[0]]
     */
    union Data {
        uint8_t bytes[4];
        uint32_t raw;
    };

```

```

    float f;
};
Data data;
void initFromFloat(float);
};

} // namespace floating

```

Użytko również dodane w C++11 user-defined literals, które zdefiniowane w następujący sposób:

```

namespace floating
{
    namespace literal
    {
        Single operator""_s(long double longDouble);
        Single operator""_s(unsigned long long uLongLong);
    } // namespace literal
} // namespace floating

```

umożliwiają tworzenie nowych obiektów w prosty sposób:

```

using namespace floating::literal;
...
floating::Single s = 1.2543_s;

```

Typ `floating::Half` od typu `floating::Single` różni się jedynie mniejszą, wewnętrzną reprezentacją, oraz literałem `_h` zamiast `_s`.

```

/**
 *          sign          fraction
 * Format:  b | bbbbbb | bbaaaaaaaa
 *          exponent
 * Half in memory looks like:
 * [b      a      ]
 * [bytes[1]bytes[0]]
 */
union Data {
    uint8_t bytes[2];
    uint16_t raw;
};

```

Testy i wydajność

Proces tworzenia biblioteki

Podczas tworzenia kolejnych elementów biblioteki zastosowana została technika *Test-Driven Development*, również podczas tworzenia operacji *Simple*. Poprzez napisany skrypt biblioteka, jak i testy, kompilowały się automatycznie po każdej zmianie kodu.

```
#!/bin/bash
{
    make ${1:-all}
} && {
    while inotifywait -r -e modify ${2:-src tests/float}; do
        make ${1:-all}
    done
}
```

Testy jednostkowe

Testy jednostkowe zostały stworzone wykorzystując framework *Catch2*. Osobno dla *floating::Single* i *floating::Half* został utworzony plik wykonywalny zawierający test każdej funkcji, które były rekompilowane i uruchamiane po każdej zmianie kodu. Środowisko *Visual Studio Code* poprzez swoje rozszerzenia integrujące *Catch2* pozwoliło na łatwe zarządzanie i wykonywanie testów.

Testy wydajności

Testy wydajności odbyły się na maszynie z systemem *Ubuntu 18.4*, na procesorze Intel(R) Core(TM) i7-4770S CPU @ 3.10GHz podczas normalnego obciążenia systemu. Pliki wykonywalne były kompilowane w architekturze 32 bit i uruchamiane z możliwie najwyższym priorytetem (*niceness=-20*). Obiekt klasy *floating::Tester* wielokrotnie wykonywał operację zdefiniowaną w przekazanej funkcji lambda (dla tych samych i różnych danych) i mierzył czas jej wykonania w cyklach pracy procesora za pomocą rozkazu *rdtsc*, który wymaga serializacji (*cpuid*):

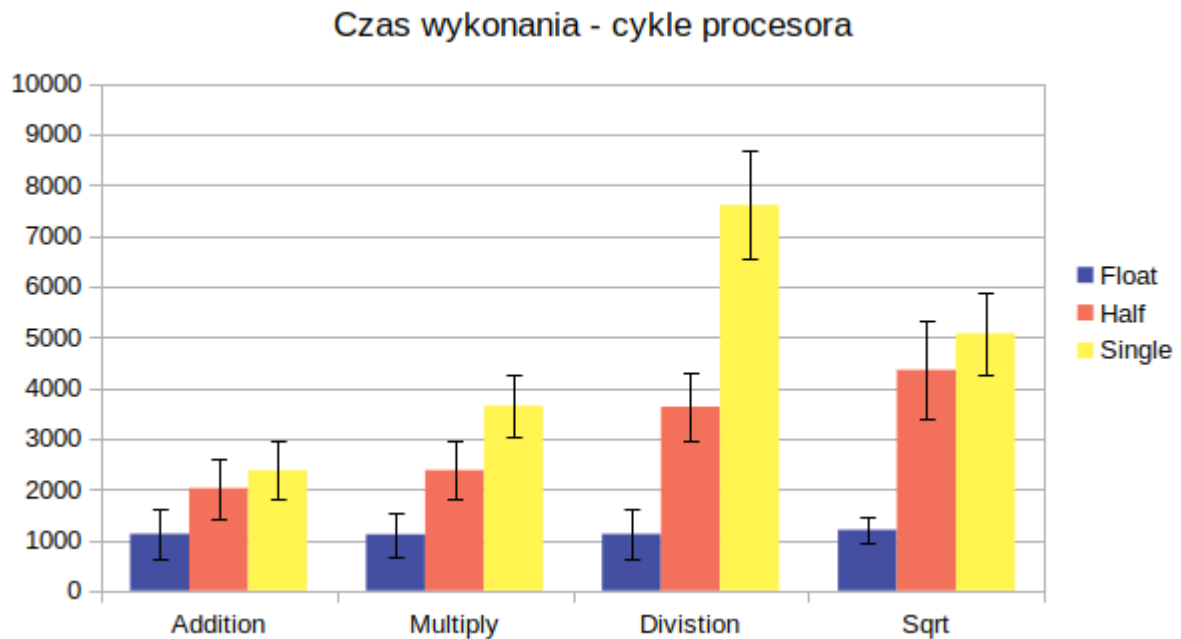
```
uint64_t rdtsc()
{
    unsigned int lo, hi;
    __asm__ __volatile__ ("cpuid; rdtsc"
                        : "=a"(lo), "=d"(hi));
    return ((uint64_t)hi << 32) | lo;
}
```

Testy każdej operacji dla argumentów z wygenerowanej przestrzeni liniowej zostały wykonane dla typów *floating::Single* i *floating::Half* oraz *float* w celu porównania do natywnej realizacji.

Wyniki testów

Wyniki ogólne

	Float		Half		Single	
	\bar{x}	σ	\bar{x}	σ	\bar{x}	σ
Addition	1119,07	504,52	2019,15	587,29	2372,03	574,55
Multiply	1107,88	427,26	2373,49	562,86	3646,37	609,94
Division	1117,67	495,26	3626,78	660,89	7603,06	1061,93
Sqrt	1198,21	249,23	4351,41	963,31	5073,23	811,81

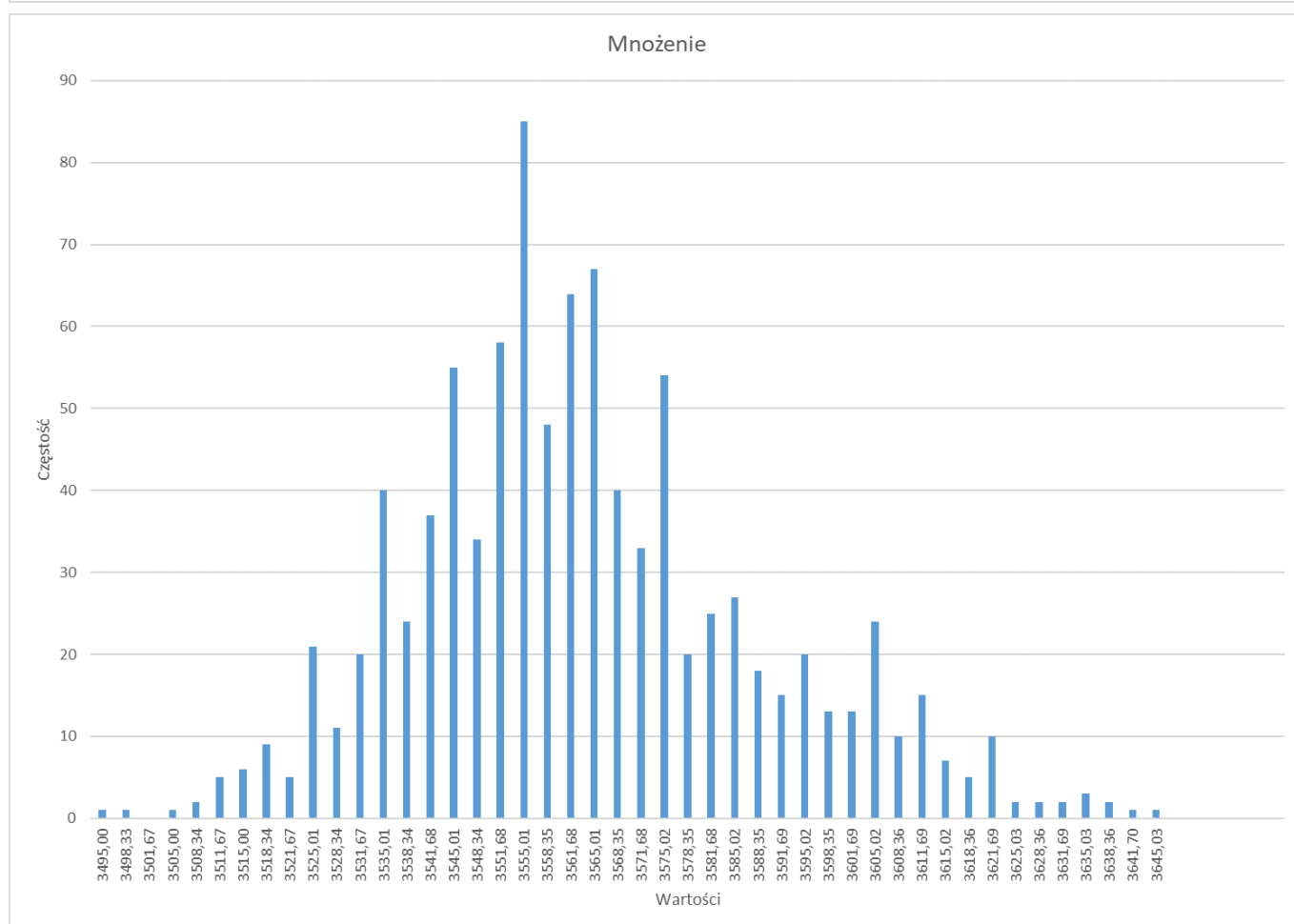
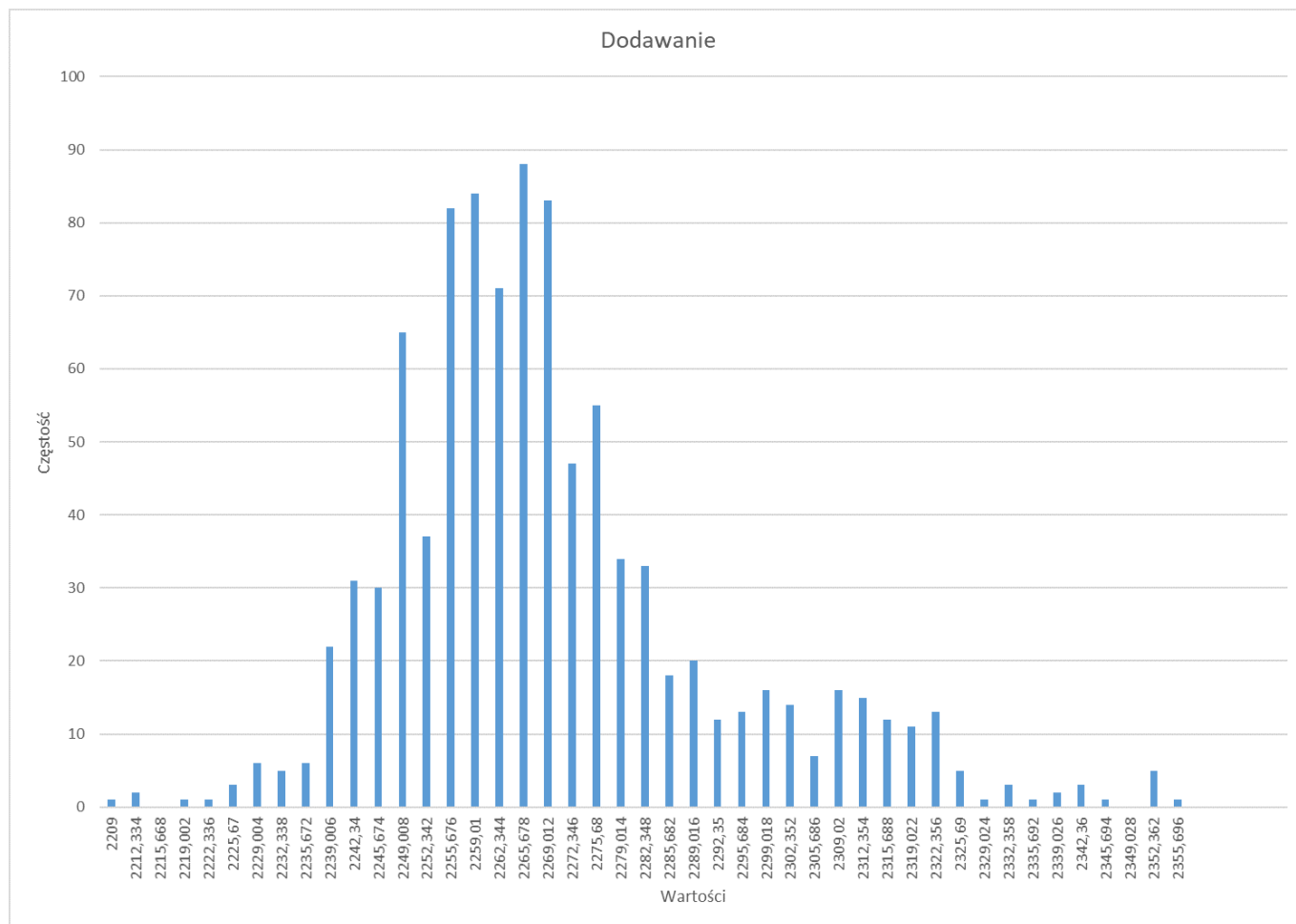


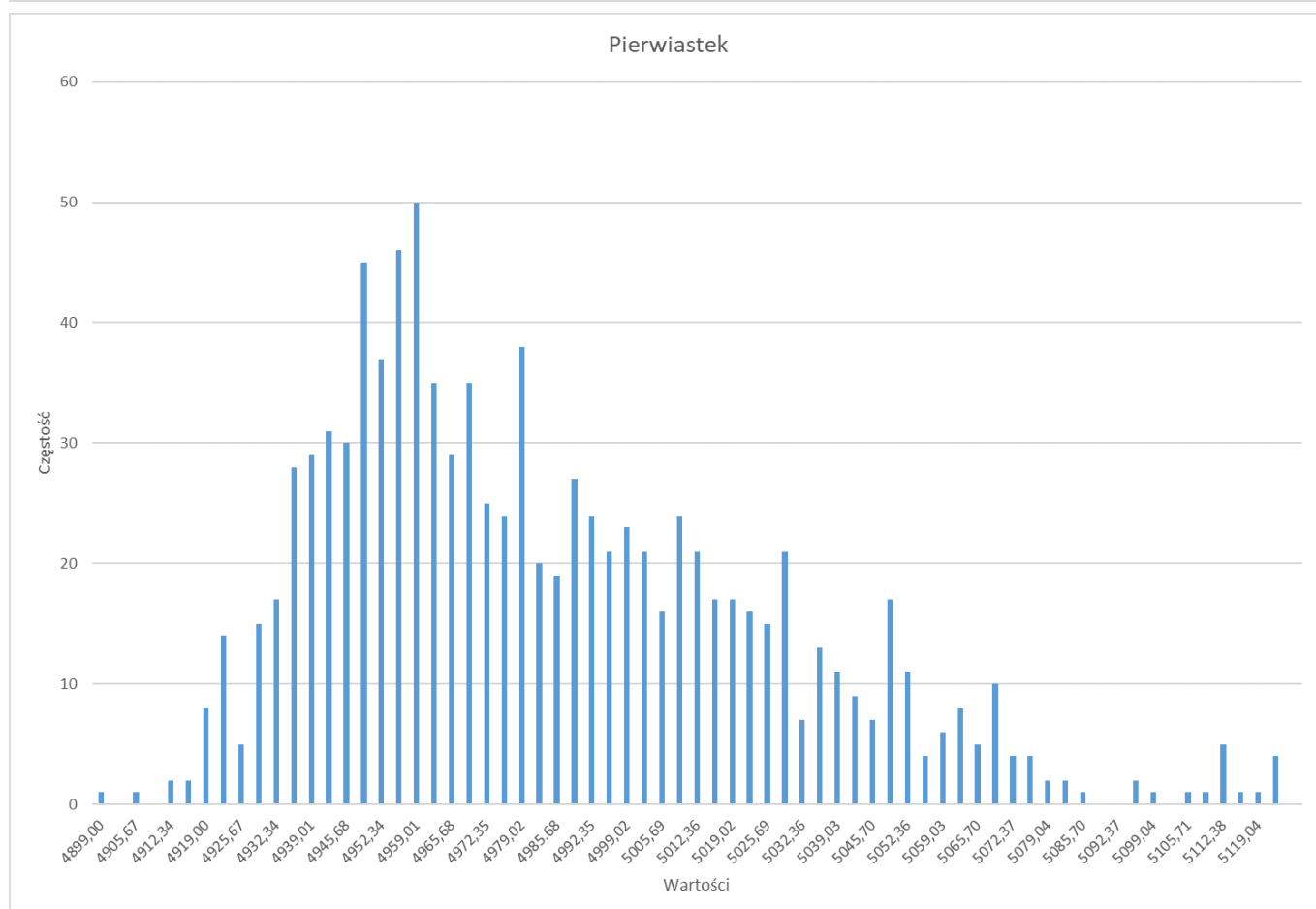
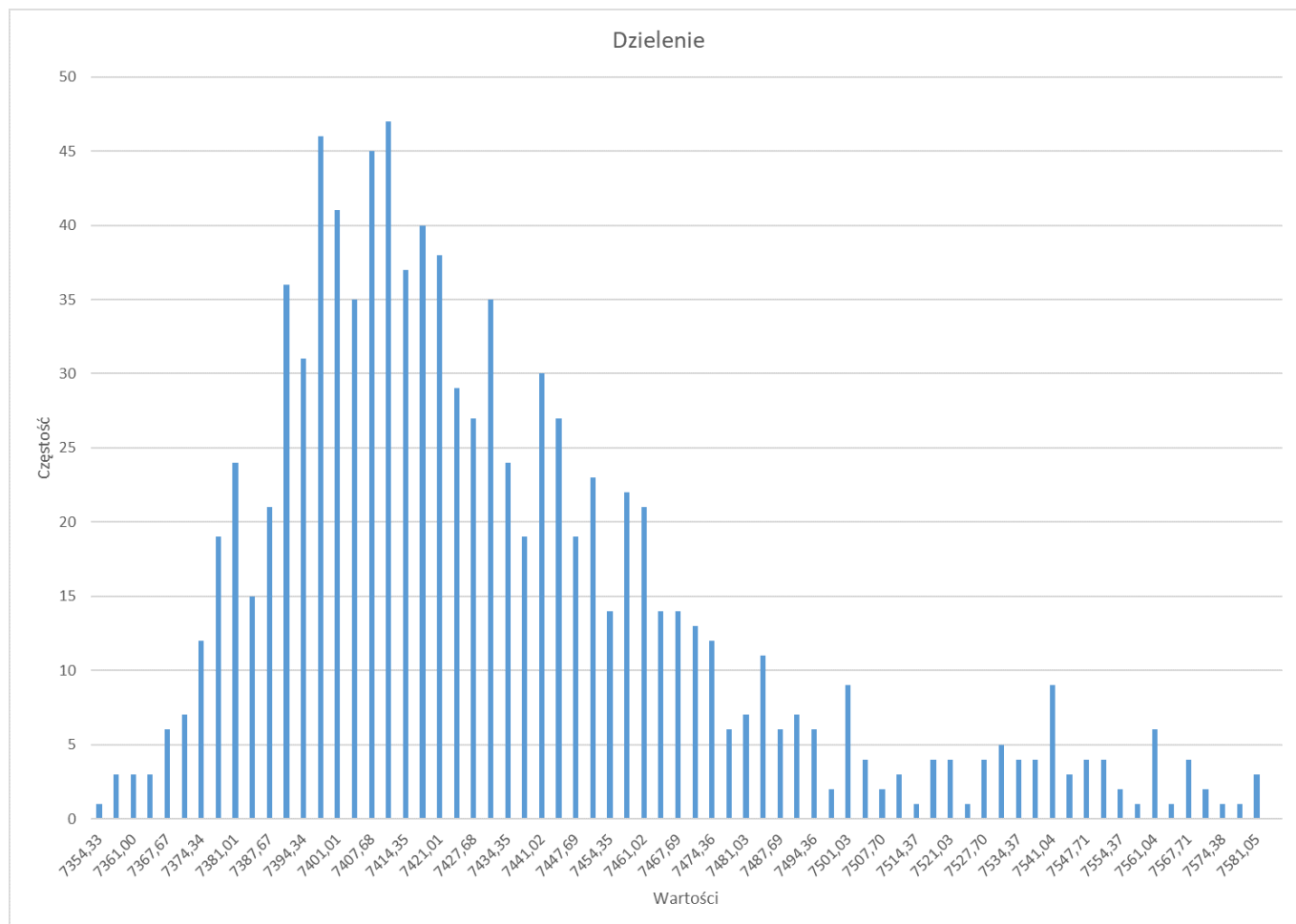
Histogramy

Dla każdej z operacji Single zostały wygenerowane histogramy. Każdy z nich przedstawia rozkład czasu trwania operacji dla 1000 powtórzeń działania dla tych samych operandów. Dla każdej operacji występują sporadycznie wartości bardzo duże, które nie zostały uwzględnione na wykresach.

Rozbieżność czasów dla tych samych operacji wynika z aktualnego obciążenia systemu operacyjnego, który zarządzając zadaniami, może przełączać kontekst pomiędzy procesami i obsługiwać przerwania. Sprawia to, że w rzeczywistości algorytmy, z punktu widzenia procesora, nie wykonują się jak jeden spójny ciąg instrukcji.

Dla operacji Half występują analogiczne rozbieżności.

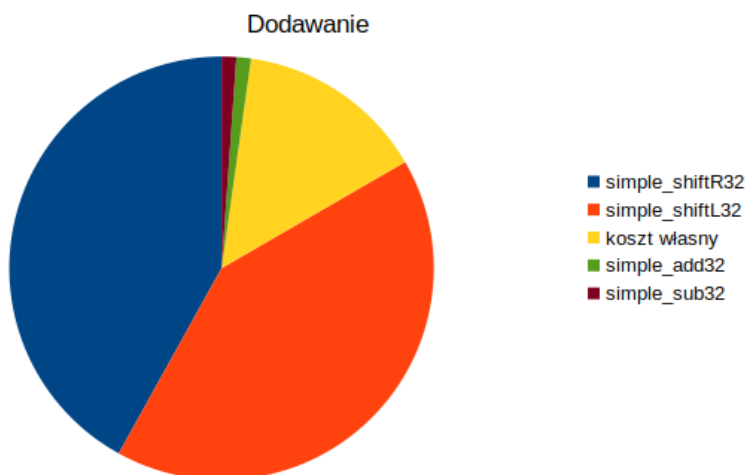




Profilowanie

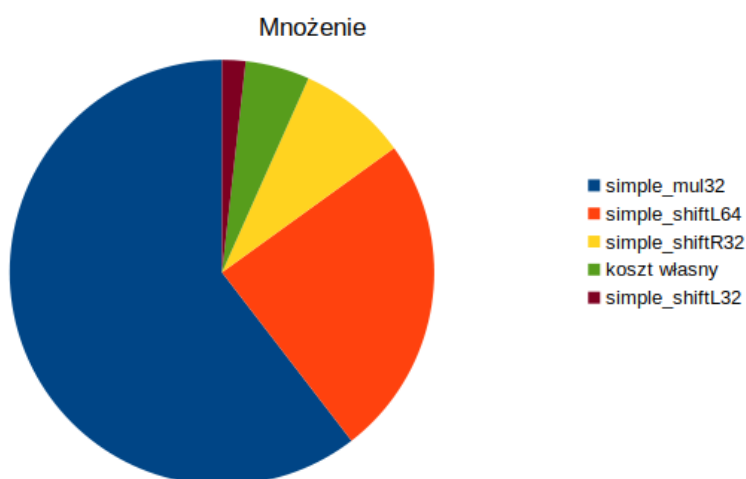
Profilowanie wykonane zostało przy użyciu *Callgrind* oraz *KCachegrind*. Na podstawie oszacowanych cykli oraz mapy wywoływanych w *KCachegrind*, udało się utworzyć wykresy reprezentujące rozkład kosztów, własnego oraz użytych w danej operacji funkcji *Simple*, w których wykonywane są operacje 8 bitowe.

Dodawanie



Większość kosztów to przesunięcia, dwa najbardziej kosztowne spośród wywołań *simple_shiftR* i *simple_shiftL* to przesunięcia, których celem jest wyzerowanie bitów znaku i wykładnika, co da się w prosty sposób wykonać przy użyciu *AND* tak, jak zostało to wykonane w implementacji dodawania dla Half. Niestety dla implementacji w *Single* zostało to przeoczone podczas optymalizowania kodu. Drugie najbardziej kosztowne wywoływania przesunięć to skalowanie mantysy wynikowej po wykonaniu operacji dodawania lub odejmowania.

Mnożenie



W przypadku mnożenia, najbardziej kosztowne jest wymnożenie mantys. Drugą najbardziej kosztowną operacją jest przesunięcie wyniku mnożenia, który maksymalnie zajmuje 48 bitów o 16 bitów w prawo. To samo dałoby się osiągnąć znacznie optymalniej przepisując odpowiednio kolejne bajty.

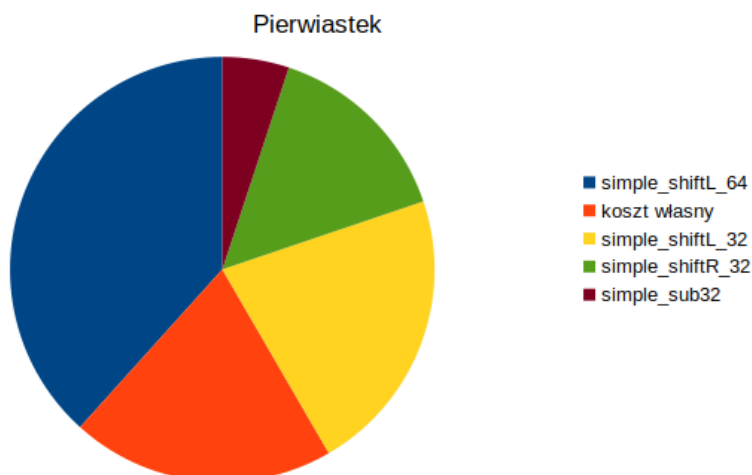
Dzielenie



Operacja `simple_div` zajmuje większą część kosztów operacji dzielenia. Ograniczenie do użycia rejestrów 8 bitowych okazało się być w przypadku algorytmu dzielenia najbardziej kosztowne.

Operacja `simple_div` w pierwszej wersji wykorzystywała algorytm dzielenia nieodtworzącego. Później został on zamieniony na algorytm wykorzystujący iteratywną korekcję błędu[1], operujący na cyfrach liczby w bazie 2^{256} , co pozwoliło wykorzystać dostępną 8 bitową operację dzielenia. Mimo zmiany algorytmu na pozornie szybszy, wydajność operacji nieznacznie się zmniejszyła.

Pierwiastek



Większość kosztów pierwiastka to przesunięcia, które wykonywane są w pętli podczas wykonywania algorytmu $(2 * Q_i * B + x) \times \leq R_i$.

Napotkane problemy

1. Brak możliwości porównania wydajności operacji z biblioteką `soft-float`. Kompilacja plików za pomocą `gcc` z flagą `-msoft-float` generuje błędy linkowania, ponieważ biblioteka `soft-float` domyślnie nie jest obecna w `libgcc`, a wszelkie próby kompilowania jej ze źródeł nie przyniosły żadnych efektów.
2. Z niezidentyfikowanych przyczyn kompilacja z optymalizacją `-O1`, `-O2`, `-O3` generuje błędy.

Wnioski

Poprawnie napisana i w całości przetestowana biblioteka realizująca obliczenia zmiennoprzecinkowe, która nie korzysta ze sprzętowego wspomaganie tych obliczeń, jest niezastąpiona na urządzeniach bez owego wspomaganie. Tworząc kod w języku assembler dla kluczowych jej elementów, ograniczamy się do obsługi jednej platformy. Kod tworzony na wyższych warstwach abstrakcji języka ma dużą zaletę, jaką jest przenośność, ponieważ to od kompilatora tego języka zależy, jaką postać kodu maszynowego wygeneruje, co pozwala na proces kompilacji tego samego kodu na różne platformy.

Spełnienie ograniczenia długości słowa 8 bit wymagało jednak stworzenia większej części kodu w języku assembler, ponieważ GCC nie pozwala na kompilację, której wynik spełniałby to ograniczenie. Podchodząc do problemu jeszcze raz, można pokusić się o stworzenie i kompilację kodu w środowisku, gdzie do testów zostałby użyty 8 bitowy mikrokontroler lub jego symulator.

Analizując opisane i przetestowane operacje i ich składowe widać, że największy narzut czasowy stanowią te, które znoszą ograniczenie długości słowa - *Simple*. Można zatem stwierdzić, że długość słowa i szybkość wykonywania na nim najprostszych operacji w dużym stopniu stanowi o szybkości działania złożonych algorytmów - w tym przypadku obliczeń na liczbach zmiennoprzecinkowych.

Kluczowym elementem poprawiającym wydajność jest również, prócz stworzenia prostych funkcji ogólnego przeznaczenia, dostosowanie ich odpowiedników do poszczególnych algorytmów w celu uniknięcia wykonywania nadmiarowej liczby kroków.

Repozytorium

<https://github.com/damiankoper/OiakProject>

Sprawozdanie

<https://github.com/damiankoper/OiakProject/blob/master/docs/Sprawozdanie.pdf>

Literatura

1. <http://justinparrtech.com/JustinParr-Tech/an-algorithm-for-arbitrary-precision-integer-division/>
2. <http://x86asm.net/articles/fixed-point-arithmetic-and-tricks/>
3. <https://github.com/lattera/glibc/tree/master/soft-fp>
4. <http://www.rfwireless-world.com/Tutorials/floating-point-tutorial.html>