CZ-TP-13 Wrocław, 16.05.2019r.

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 | xxxxxxxxxxxxxxxxxxxxxxx
    wykładnik
```

Ułamek zapisny jest z dodanym obciążeniem, co pozwala zachować ciągłość reprezentacji i ułatwia porównania. Dla formatu single obiążenie 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 | xxxxxxxxxx
    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 przemiennym, warto rozpatrywać zawsze jeden przypadek, gdzie *A <= B*. W przypadku kiedy *A > B* należy zamienić oba składniki miejscami.

Następnie trzeba wyrównać wykładnik mniejszej liczby. Wiemy, że *A <= B*, więc musimy przesunąć bity mantysy B o `exp[A]-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 wartośc `exp[A]`. Przy wynikach, dla których wartość nie mieści sie 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 `sign[C] = sign[A] XOR sign[B]`. Następnie sprawdzamy, czy którykolwiek ze składników ma wartośc *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 obiążenia, 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 wartośc 1 oznacza to, że wartość jest za duża. Trzeba zwiększyć wykładnik i przesująć mantysę w prawo.

W przypadku, gdy wartość jest zbyt mała, trzeba sprawdzać bit 47 iloczynu, zmniejszać wykładnik i przesuwać mantysę w lewo, aż owy bit nie będzie miał wartości 1 lub wykładnik nie będzie równy `0x01` (dla Half `0b00001`) - w takim wypadku otrzymamy wynik zdenormalizowany. W przypadku wyniku zdenormalizowanego wykładnik reprezentowany jest jako `0x00` (dla Half `0b00000`).

Zaraz po wykonaniu mnożenia mantys, znając pozostałe bity wyniku, które zostaną utracone, możemy wykonać zaokrąglanie. Tak jak w dodawaniu/odejmowaniu, w przypadku przekroczenia zakresu musimy pamiętać o ustawieniu wartości *+/-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 *Q13*) 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 | mod(frac[A]/frac[B]) != 0
```

uwzględnia bity reszty.

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

W przypadku przekroczenia zakresu musimy pamiętać o ustawieniu wartości *+/-0*.

Pierwiastek

Pierwiastkowanie możliwe jest tylko gdy liczba jest dodatnia więc rozpoczynamy 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 * Qi * B + x)x <= Ri` , gdzie Qi – aktualny wynik, B – podstawa liczby, w tym przypadku 2, Ri – aktualna reszta, x to kolejny bit wyniku pierwiastkowania, w przypadku pierwiastkowania liczb o podstawie 2, x równe jest 1 jeśli dla x równego 1 spełnione jest 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 zostały zrealizowane poprzez stworzenie biblioteki języka `C++`. Architektura projektu została zrealizowana w trzech częściach, gdzie każda korzystała 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 dłogości słowa. Używają one instrukcji operujących na liczbach 8 bitowych, wykorzystujących flagę przeniesienia oraz algorytmy m.in dzielenia nieodtwarzającego.

Single/Half

Część *Single/Half* implementuje obliczenia na liczbach zmiennoprzecinkowych połowicznej i pojedynczej precyzji używając operacji *Simple*. Wegług 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 liczby na różne sposoby.

```
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 | ddddddddc | cccccccbbbbbbbbaaaaaaaa
   *              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 spodób:

```
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 | bbbbb | 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

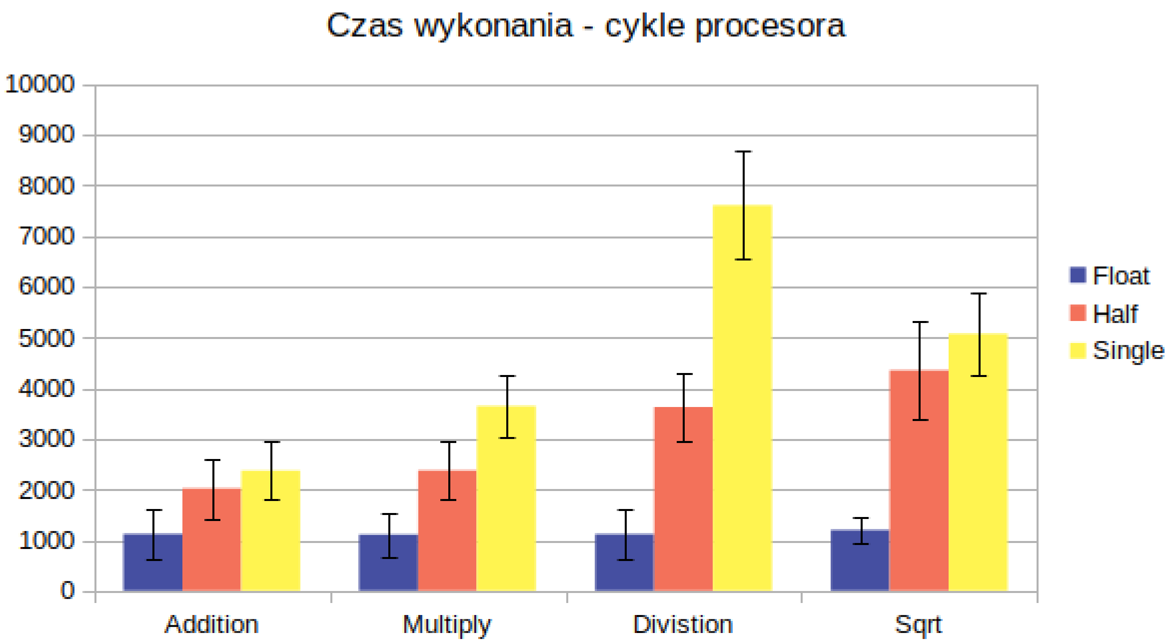
Testy wydajności odbyły się na maszynie z systemem *Ubuntu 18.4*, na procesorze Intel(R) Core(TM) i7-4770S CPU @ 3.10GHz podczas normalnego obciążenia systemu. Pliki wykonywalne były kompilowane w architekturze 32 bit i uruchamiane z możliwie najwyższym priorytetem (`niceness=-20`). Obiekt klasy `floating::Tester`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

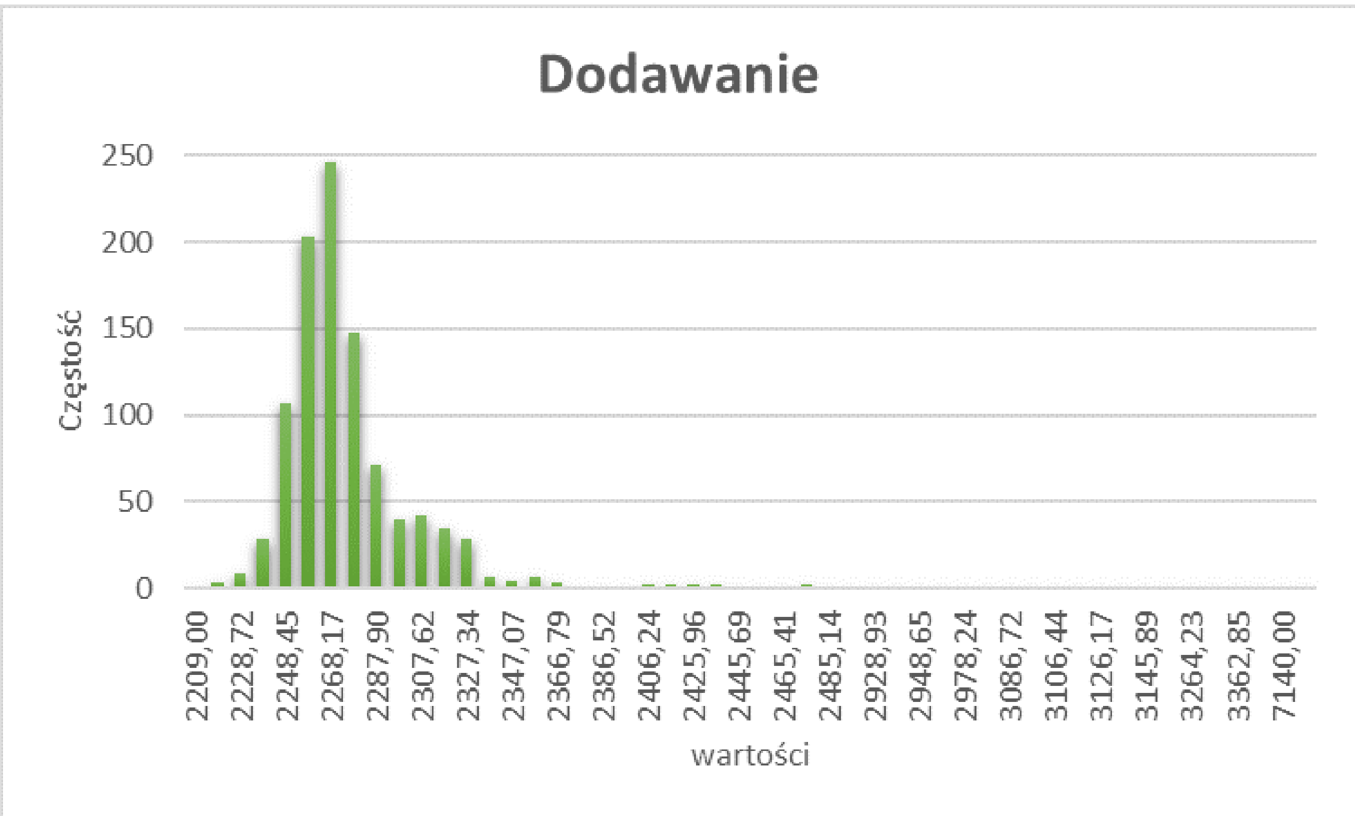
Testy każdej operacji dla argumentów z wygenerowanej przestrzeni liniowej zostały wykonany dla typów `floating::Single` i `floating::Half` oraz `float` w celu porównania do natywnej realizacji.

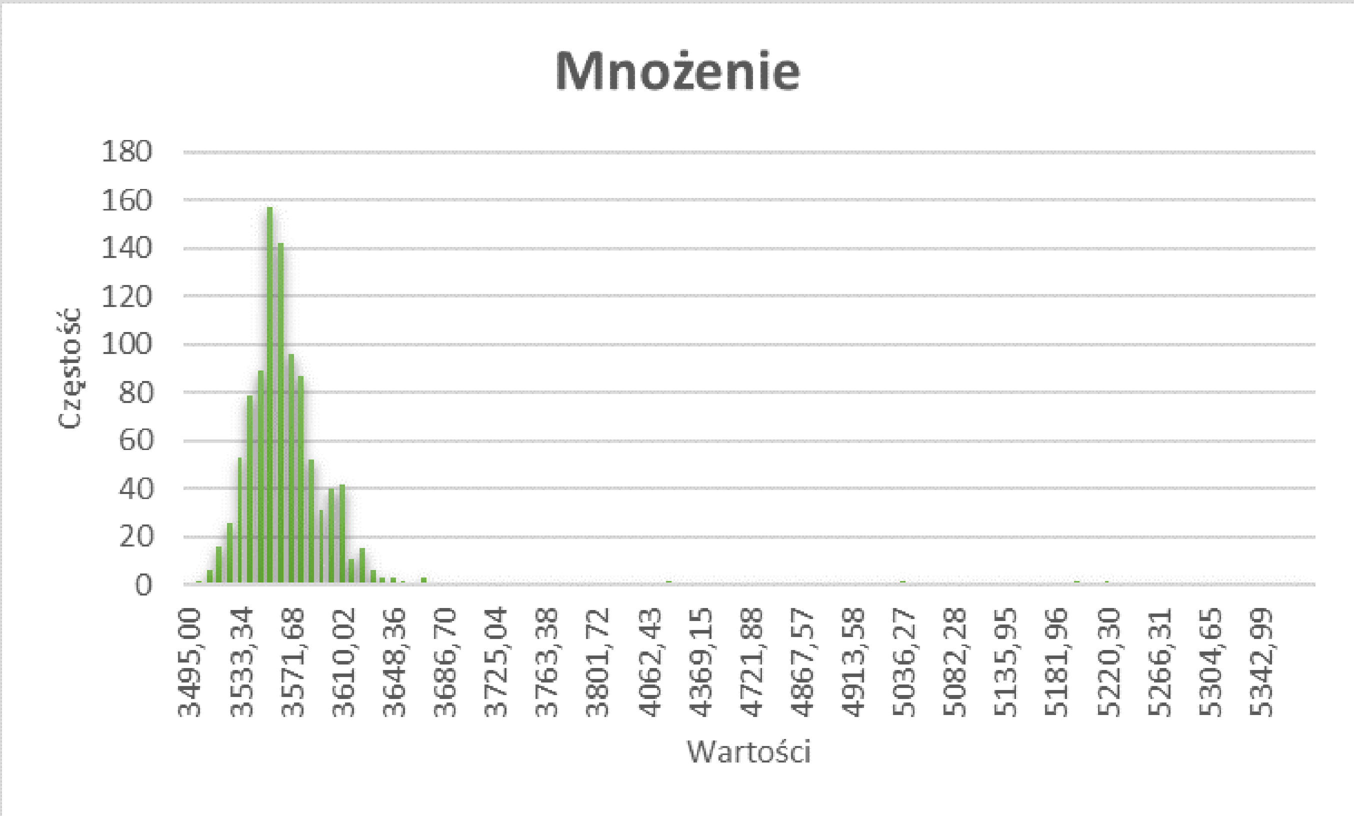
Wyniki testów

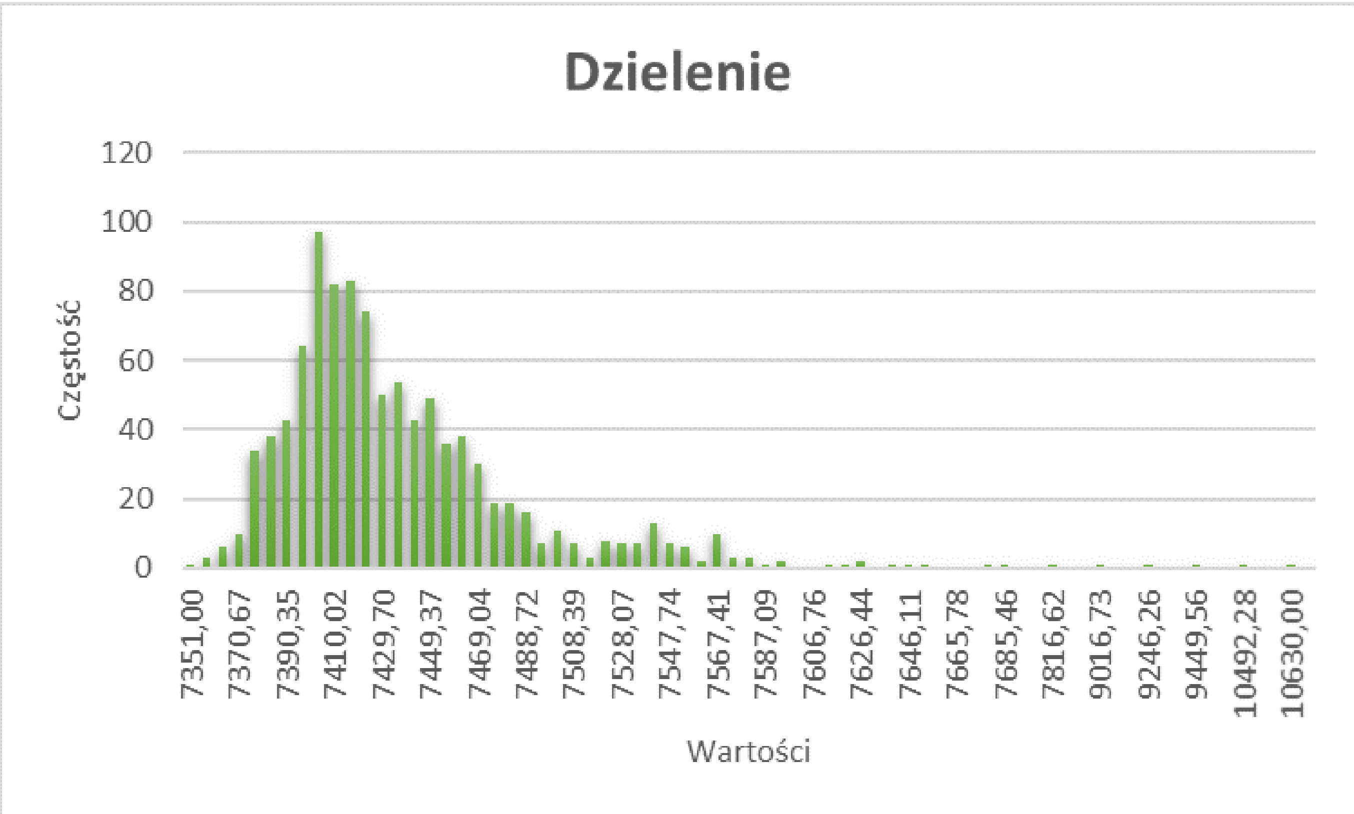
	Float		Half		Single	
	x̄	σ	x̄	σ	x̄	σ
Addition	1119,07	504,52	2019,15	587,29	2372,03	574,55
Multiply	1107,88	427,26	2373,49	562,86	3646,37	609,94
Division	1117,67	495,26	3626,78	660,89	7603,06	1061,93
Sqrt	1198,21	249,23	4351,41	963,31	5073,23	811,81

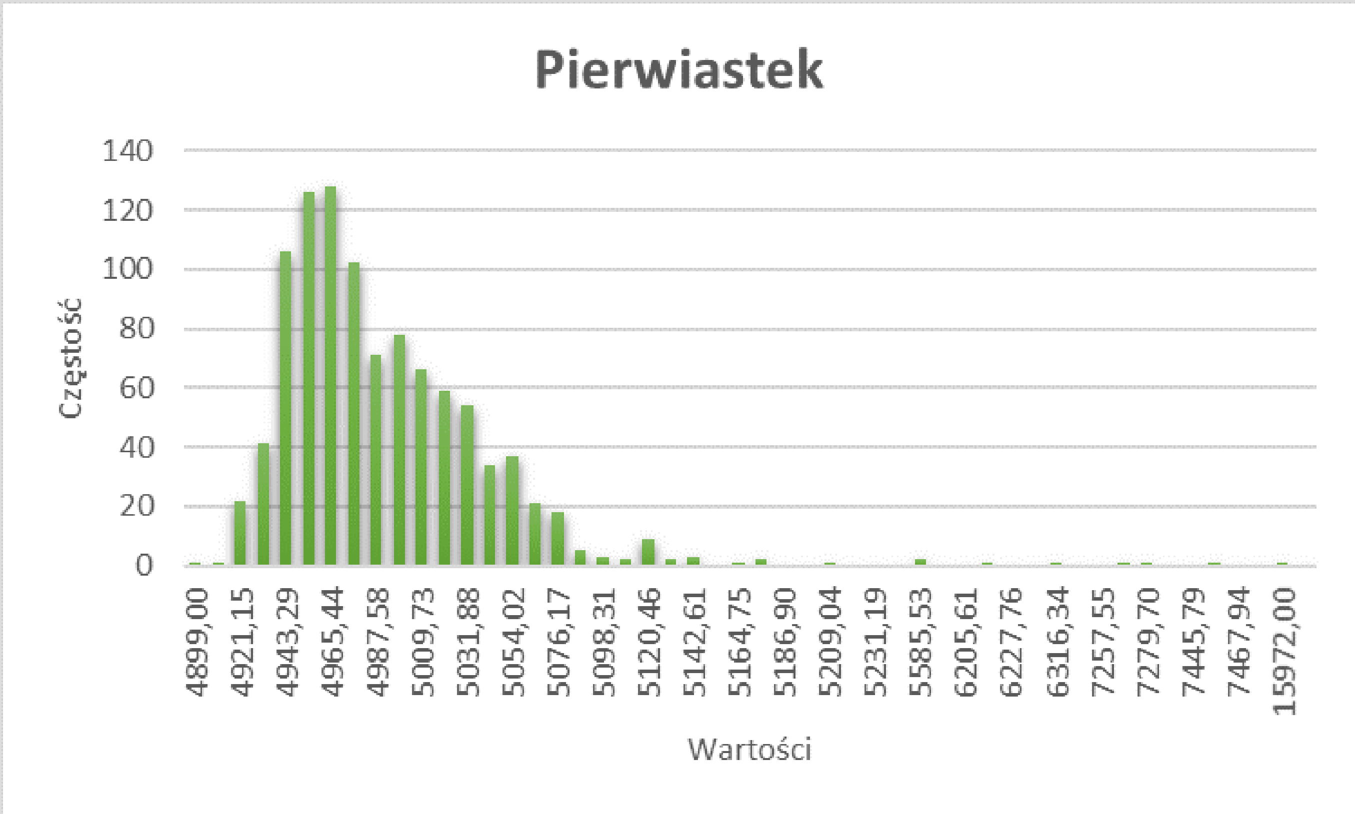


Zostały wykonane histogramy dla każdej z operacji single, dla 1000 powtórzeń na tych samych danych. Przedział wartości przedstawiony na wykresie zaczyna się od minimalnej występującej w teście wartości a kończy na maksymalnej. Do momentu w którym zaczynają występować sporadycznie duże wartości przedział zmienia się o stałą wartość, potem przerwy gdzie nie występowały długo żadne wartości wycięte są odpowiednio tak by zawrzeć na wykresie wszystkie wartości, które wystąpiły w teście i nie był on jednocześnie zbyt szeroki.

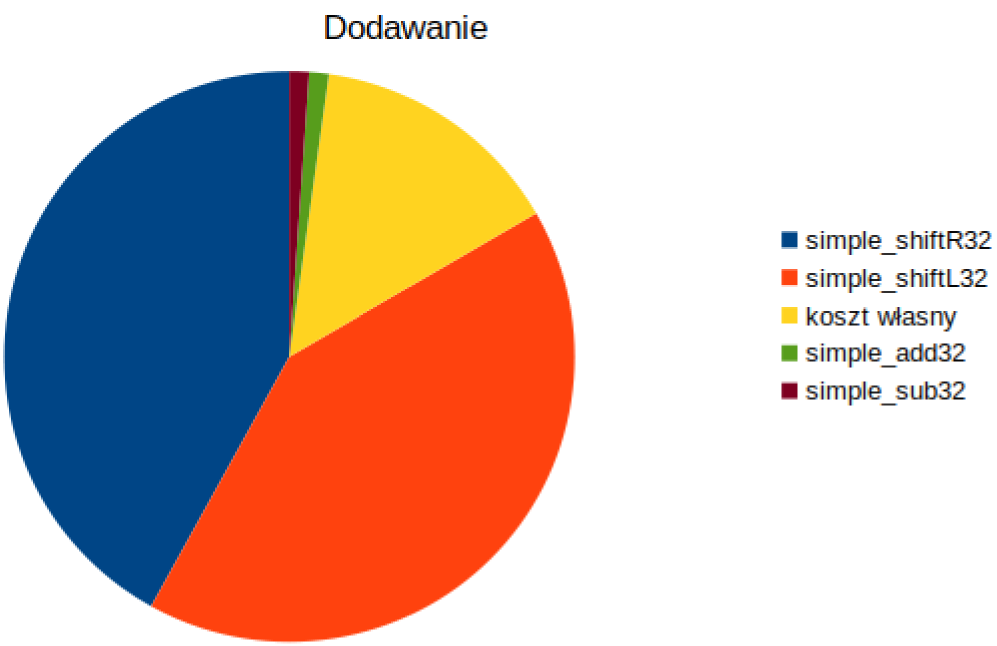



Mnożenie
180
160
140
120
100
80
60
40
20
0
Częstość
3495,00
3533,34
3571,68
3610,02
3648,36
3686,70
3725,04
3763,38
3801,72
4062,43
4369,15
4721,88
4867,57
4913,58
5036,27
5082,28
5135,95
5181,96
5220,30
5266,31
5304,65
5342,99
Wartości


Dzielenie
120
100
80
60
40
20
0
Częstość
7351,00
7370,67
7390,35
7410,02
7429,70
7449,37
7469,04
7488,72
7508,39
7528,07
7547,74
7567,41
7587,09
7606,76
7626,44
7646,11
7665,78
7685,46
7816,62
9016,73
9246,26
9449,56
10492,28
10630,00
Wartości

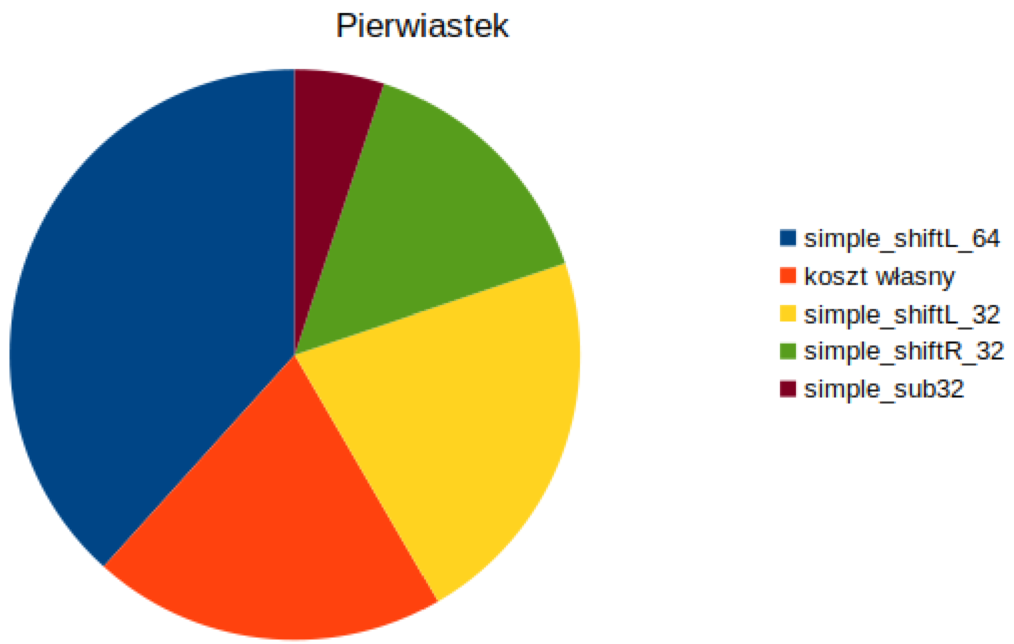


Profilowanie wykonane zostało przy użyciu Callgrind oraz KCachegrind. Na podstawie oszacowanych cykli oraz mapy wywoływanych w KCachegrind, udało się utworzyć wykresy reprezentujące rozkład kosztów, własnego oraz użytych w danej operacji funkcji simple, w których wykonywane są operacje 8 bitowe.

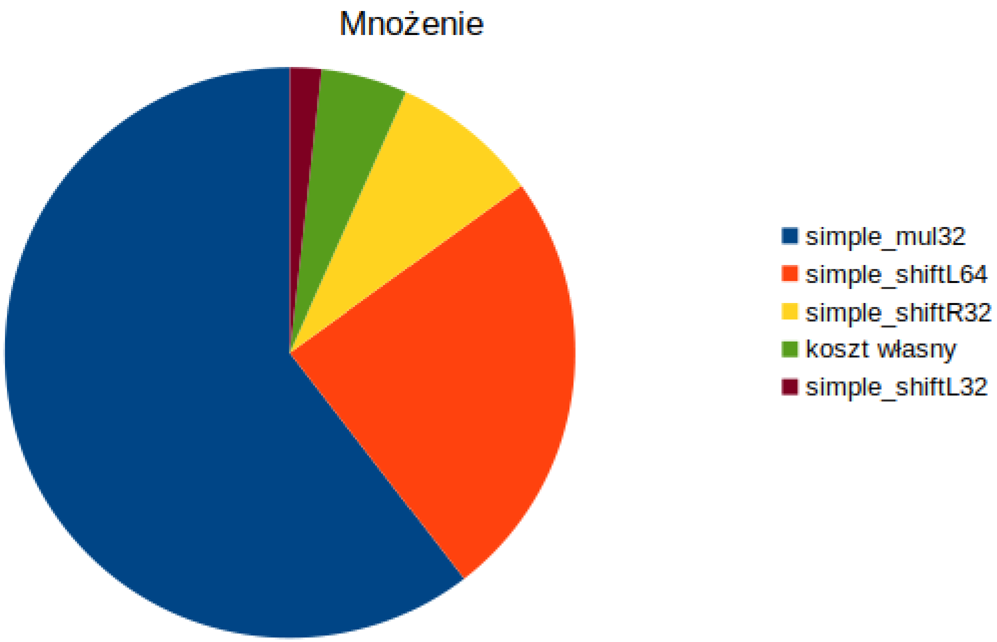


Większość kosztów to przesunięcia, dwa najbardziej kosztowne spośród wywołań simple_shiftR i simple_shiftL to przesunięcia, których celem jest wyzerowanie bitów znaku i wykładnika co da się w prosty sposób wykonać przy użyciu AND tak jak zostało to wykonane w implementacji dodawania dla half. Niestety 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.

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



Większość kosztów pierwiastka to przesunięcia, które wykonywane są w pętli podczas wykonywania algorytmu `(2 * Qi * B + x)x <= Ri`.



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 lewo. To samo dałoby się osiągnąć znacznie optymalniej przepisując odpowiednio kolejne bajty.

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

//TODO

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