Implementacja procedur obliczeń na liczbach zmiennoprzecinkowych za pomocą instrukcji stałoprzecinkowych.

Adam Krysiak 218356, Maciej Lewiński 209868

19 maja 2016

Spis treści

1 Główne założenia projektu

Celem projektu było zaimplementowanie podstawowych operacji arytmetycznych na liczbach zmiennoprzecinkowych za pomocą instrukcji stałoprzecinkowych. W ramach projektu zarealizowaliśmy trzy operacje jakimi są mnożenie, dzielenie oraz dodawanie/odejmowanie, w formacie 32bitowym IEEE-754 binary32. Biblioteka zaimplementowano jako klasa C++ o nazwie FPU, z własnymi metodami realizującymi operacje arytmetyczne. Klasa FPU posiada także zmienną rounding, która oznacza sposób zaokrąglania.

```
class FPU
{
        Rounding rounding;

public:
        FPU();
        int fadd(float, float);
        int fmul(float, float);
        int fdiv(float, float);

        void setRounding(Rounding round);
        Rounding getRounding();

        virtual ~FPU();
};
```

Listing 1: Klasa FPU

Funkcje przyjmują po dwa argumenty typu float i zwracają typ int zawierający reprezentację liczby zmiennoprzecinkowej. By uzyskać z nich faktyczny wynik, należy odpowiednio rzutować zwracaną liczbę na float.

```
*reinterpret_cast<float*>(&result)
```

Listing 2: Przykładowy sposób zamiany wyniku typu int na float

Przy włączeniu optymalizacji na poziomie O3 oraz takim rzutowaniu kompilator ostrzega nas o złamaniu zasady strict-aliasing[1]. Ponieważ robimy to świadomie, aby skompilować program bez ostrzeżeń musimy przy kompilacji dodać flagę -fno-strict-aliasing.

2 Zrealizowane operacje arytmetyczne

2.1 Dodawanie i odejmowanie

Dodawanie oraz odejmowanie zrealizowaliśmy za pomocą jednej instrukcji fadd, przyjmującej dwie liczby typu float. Przenosimy odpowiednie bity znaków, cech i mantys do odpowiadającym im zmiennych w celu ułatwienia późniejszych działań. Następnie sprawdzamy czy otrzymane liczby są bardzo małe, bardzo duże, nie są liczbami bądź są zdenormalizowane, w każdym przypadku sygnalizujemy odpowiedni wyjątek. Dodawanie wykonujemy korzystając z następującej zależności

$x_1 = M_1 * B^{E_1}$
$x_2 = M_2 * B^{E_2}$
gdzie: $E_1 > E_2$

$$x_1 \pm x_2 = M_1 * B^{E_1} + M_2 * B^{E_2} = (M_1 * B^{E_1 - E_2} + M_2) * B^{E_2}$$

Następnie dodawanie zrealizowano zależnie od znaków operandów. Następnie obliczamy różnicę między cechami obydwu operandów, w przypadku dwóch liczb dodatnich, przesuwamy mantysę pierwszego o tą różnicę, w przypadku dwóch liczb ujemnych, przesuwamy mantysą argumentu drugiego. Następnie dodajemy mantysy obydwu liczb.
$(M_1 * B^{E_1 - E_2} + M_2)$ Na tym etapie eksponentą będzie E_2. Jeżeli znaki są różne, to operacja dodawania staje się operacją odejmowania. Operacja odejmowania wygląda podobnie, jednak znak wynikowy jest znakiem argumentu o większej wartości bezwzględnej. Wynik powyższych operacji wymaga znormalizowania. W tym celu zliczana jest ilość bitów przesunięcia wymaganego do zmieszczenia liczby w formacie single precision. Ta sama liczba jest następnie dodawana do cechy wyniku. Przesunięcie jest powiększone o 3 bity ze wzlędu na bity GRS potrzebne do zaokrąglenia. Zaokrąglając przesuwamy 3 ostatnie bity, a następnie sprawdzamy czy otrzymany wynik nie jest bardzo mało, bardzo duży bądź zdenormalizowany. Jeśli tak się dzieje to sy-

[1]Strict aliasing jest założeniem, robionym przez kompilator wyłuskane wskaźniki na objekty innych typów nie będą odnosiły się do tego samego miejsca w pamięci

gnalizujemy odpowiednie wyjątki, w przeciwnym wypadku składamy wynik i zwracamy go jako zmienną typu int.

2.2 Mnożenie

Mnożenie liczb zmiennoprzecinkowych realizuje funkcja fmul. Opiera się ona na zależności:

$x_1 = M_1 * B^{E_1}$
$x_2 = M_2 * B^{E_2}$

$x_1 \pm x_2 = (M_1 * B^{E_1}) * (M_2 * B^{E_2}) = (M_1 * M_2) * B^{E_1+E_2}$

Zaczynamy od przeniesienia mantysy, cechy oraz znaku do osobnych zmiennych, na których wykonywać będziemy operacje. Następnie sprawdzamy czy liczba jest zdenormalizowana, bardzo mała, bardzo duża lub NaN'em, w każdym z tych przypadków rzucamy odpowiedni wyjątek. Znak wynikowy uzyskujemy poprzez wykonanie operacji xor na znakach obydwu operandów. Cechy dodajemy do siebie ($E_1 + E_2$), a następnie odejmujemy od wyniku 127 ze względu na obciążenie. Mantysy rzutujemy na typ long long, mnożymy ($M_1 * M_2$) i zapisujemy w zmiennej o tym samym typie. Zastosowaliśmy long long ze względu na ilość potrzebnych bitów do zapisania wyniku mnożenia. Następnie sprawdzamy, czy wynik zmieści się w wymaganych 23 bitach formatu single precision. Jeśli nie, to zapisujemy bity RS i na ich podstawie zaokrąglamy wynik, a następnie przesuwamy o jedną pozycję w prawo, następnie, niezależnie od tego czy wynik był normalizowany czy nie, przesuwany jest on o 23 bity w prawo w celu późniejszego złożenia wyniku. Na końcu ponownie sprawdzamy czy uzyskane wyniki są poprawne, jeśli nie to rzucamy odpowiedni wyjątek, jeśli tak, to składamy cały wynik za pomocą operatorów or i zwracamy go w zmiennej int.

2.3 Dzielenie

Za operację dzielenia odpowiada funkcja fdiv i została ona wykonana analogicznie do operacji mnożenia, jednak w tym przypadku, cechy odejmujemy od siebie, natomiast mantysy dzielimy. Sprawdzany jest także dodatkowo w stosunku do mnożenia, czy wystąpiło dzielenie przez zero, jeśli tak, to rzucany jest odpowiedni wyjątek. W przypadku dzielenia mantys, ważnym jest, by przed wykonaniem operacji, przesunąć je na wyższe bity, a potem z powrotem wyrównać tak by nie utracić bitów wyniku. W przypadku zaokrąglaniu

dzielenia musimy wykorzystać także dodatkowy bit G. Normalizacja w przypadku dzielenia odbywa się w nieco inny sposób. W pętli while zliczana jest ilość bitów, o którą musimy przesunąć wynik w celu normalizacji oraz dodać do cechy. Podobnie jak w przypadku mnożenia, po normalizacji sprawdzamy odpowiednie wyjątki, a następnie składamy wynik i zwracamy go.

3 Zaokrąglanie

Poprzednio opisane operacje implementują także system zaokrąglania uzyskanych wyników. Ze względu na to, że operacje te, dają w wyniku zwykle więcej bitów niż jesteśmy w stanie pomieścić w formacie single precision, musimy wyniki normalizować. Zaokrąglanie polega na uwzględnieniu bitów GRS (jedynie RS w przypadku mnożenia) i na ich podstawie zaokrąglany jest wynik przy normalizacji. Dla koprocesora domyślnym sposobem zaokrąglania jest zaokrąglanie symetryczne do parzystej. W pracy zaimplementowaliśmy trzy rodzaje zaokrąglania:

1. Symetryczne do parzystej(NEAREST)
2. Do minus nieskończoności(MINUS_INF)
3. Do plus nieskończoności(PLUS_INF)

```
enum class Rounding
{
        MINUS_INF = 0,
        NEAREST = 1,
        PLUS_INF = 2
};
```

Listing 3: Enum zawierający tryby zaokrąglania

Koprocesor ma także opcję zaokrąglania do zera (przez tzw. obcięcie). Trybu tego nie zrealizowaliśmy w naszej implementacji. W celu ustawienia zaokrąglenia, należy wywołać metodę klasy FPU o nazwie setRounding(), która jako parametr przyjmuje enum z powyższymi wartościami.

```
my_fpu.setRounding(Rounding::NEAREST);
```

Listing 4: Przykładowa zmiana zaokrąglania w klasie FPU

Domyślnie zaokrąglanie ustawione jest na NEAREST.

4 Obsługa wyjątków

Wyjątki rzucane są gdy otrzymane operandy są nieprawidłowe bądź kiedy wynik danej operacji jest nieprawidłowy. Zaimplementowane wyjątki znajdują się w pliku nagłówkowym FPUExceptions.h. Każdy z wyjątków zwraca jedynie łancuch znaków informujący o tym co się stało, co na potrzeby tej pracy było wystarczającym rozwiązaniem. Zaimplementowano następujące wyjątki:

1. Nie liczba - FPU_NAN_Exception
2. Plus nieskończoność - FPU_plusInf_Exception
3. Minus nieskończoność - FPU_minInf_Exception
4. Liczba zdenormalizowana - FPU_Denormalized_Exception
5. Dzielenie przez zero - FPU_divideByZero_Exception

Wyjątki zrealizowano za pomocą specjalnie utworzonych klas FPU Exception:

```
class FPU_NAN_Exception: public std::exception
    {
    public:
    virtual const char* what() const throw ()
        {
        return "Not a number Exception occured";
        }
    };
```

Listing 5: Implementacja przykładowego wyjątku rzucanego przez operacje arytmetyczne

Reszta wyjątków została utworzona analogicznie do powyższego. Jak widać, wszystkie wyjątki dziedziczą po klasie std::exception dzięki czemu łapiąc taki wyjątek w programie jesteśmy w stanie wywołać dla niego funkcję what() by wyświetliś stosowny komunikat.

5 Testy jednostkowe

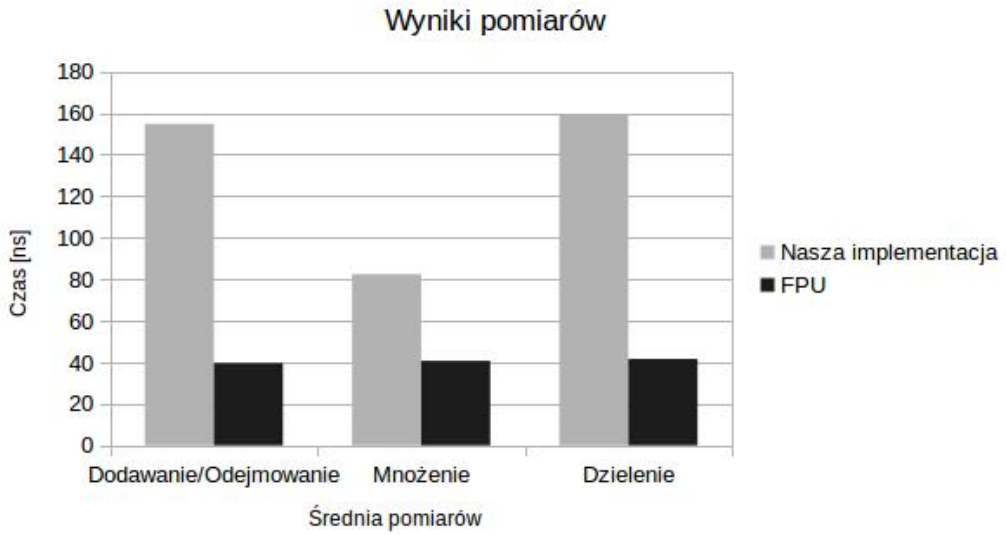
Testy robione były przy pomocy biblioteki Google Test. Testy zostały napisane w taki sposób, by sprawdzały wyrzucanie odpowiednich wyjątków oraz działania 'normalne' dla danej operacji sprawdzające każdy z trzech trybów zaokrąglania. Sprawdzenia rzucanych wyjątków dokonujemy przekazując do funkcji wartości, które powinny spowodować rzucenie wyjątku w wyniku lub już na podstawie analizy odebranych argumentów. Następnie w klauzuli catch łapiemy stosowny dla danego testu wyjątek. Jeśli wyjątek został złapany, test jest akceptowany. Testy zaokrąglenia zostały napisane tak by faktycznie było widać różnicę w trybach zaokrąglania. Dla operacji fadd podaliśmy takie liczby, które wygenerują każdą możliwą kombinację bitów GRS by pokazać, że zaokrąglanie działa w każdym przypadku. Dla pozostałych operacji przekazaliśmy tylko liczby, dla których będzie widać różnicę w wynikach. W testach wykorzystana została funkcja ASSERT_EQ zamiast ASSERT_FLOAT_EQ, który domyślnie ignoruje ostatnie bity ze względu na zaokrąglenia. Przed samymi testami ustawiany był tryb zaokraglania poprzez specjalnie napisanego na tą potrzebę podprogramu w Assemblerze x86 do kontroli FCTRL.

6 Porównanie wydajności

Czas wykonywanych operacji mierzony był za pomocą biblioteki Chrono, a wszystkie operacje zostały powtórzone w pętli dziesięć razy. Pomiary wykonywany były na laptopie HP Envy o procesorze:

1. Intel Core i7 4702MQ @ 2.2GHz

Ze względu na ograniczony dostęp do BIOSu, w laptopie nie udało się wyłączyć na czas testów technologii Intel Speedstep, zmieniającej częstotliwość taktowania CPU w zależności od wymagań. W wynikach pominięto pierwszych pięć pomiarów, które zawsze były wiele razy wolniejsze ze względu na mechanizmy pracy procesora. Resztę pomiarów uśredniono oraz policzono odchylenie standardowe dla każdej operacji.



Rysunek 1: Wykres pokazujący uśrednione wyniki pomiarów

Tabela 1: Uśrednione wyniki pomiarów wraz z odchyleniem standardowym

	Nasza implementacja		FPU	
	Średnia [ns]	Odchylenie standardowe [ns]	Średnia [ns]	Odchylenie standardowe [ns]
Dodawanie i odejmowanie	154.74	13.39	39.59	4.43
Mnożenie	82.39	10.03	40.67	3.88
Dzielenie	159.59	12.76	41.61	5.65

Tak jak można się było spodziewać, nie sposób dorównać prędkością do funkcji natywnych koprocesora, jednak wyniki uzyskane przez naszą implementację są zadowalające, nawet pomimo tego, że odchylenie standardowe jest nieco większe od tego dla funkcji natywnych przez co czas wykonywania ma większy rozrzut.

7 Podsumowanie

W ramach pracy zaimplementowano klasę FPU, pozwalająca wykonywać podstawowe operacje (dodawanie, odejmowanie, mnożenie i dzielenie) na liczbach zmiennoprzecinkowych jedynie za pomocą instrukcji stałoprzecinkowych. Wszystkie zaimplementowane operacje działają i zostały przetesto-

wane za pomocą biblioteki Google Test. Klasa FPU obsługuje także 5 typów wyjątków rzucanych w omówionych przypadkach.

8 Źródła

[1] dr. inż. Tadeusz Tomczak, Seria wykładów z Arytmetyki komputerów, wykłady 4 i 5, Wrocław, 2015.

[2] Niezależne językowo forum pytań i odpowiedzi dla programistów http://stackoverflow.com/ (wiele pytań)

[3] Serwer studencki Wydziału Elektroniki i Technik Informacyjnych http://home.elka.pw.edu.pl/ pmodlins/files/gik_mnu_lab01_zmiennoprzecinkowe.pdf (dostęp 23.04.2016)

[4] Encyklopedia internetowa Wikipedia https://en.wikipedia.org/wiki/IEEE_floating_poi (dostęp 30.04.2016)

[5] Repozytorium googletest na www.github.com https://github.com/google/googletest (dostęp 12.05.2016)

[6] Dokumentacja języka C++ http://en.cppreference.com/w/ (dostęp 12.05.2016)