

Rozdział 15

Obliczania numeryczne stałoprzecinkowe

Streszczenie Obliczenia stałoprzecinkowe: cel, podstawowe operacje, ograniczenia, przykłady algorytmów, procesory sygnałowe.

15.1 Wprowadzenie - komputerowe reprezentacje liczb

Obecnie zrobimy krótkie przypomnienie informacji z pierwszego wykładu dotyczące liczb zmiennoprecinkowych. Poniżej podano wzory na wartości tych liczb (dla liczb zmiennoprecinkowych: E - eksponenta/wykładnik, M - mantysa/cecha):

- zmiennoprecinkowa **float** - 32-bitowa (S - 1 bit, E - 8 bitów, M - 23 bity):

$$x = (-1)^S \cdot M \cdot 2^{(E-127)}, \quad S \in \{0, 1\}, E = 0, 1, \dots, 255, M = 1 + b_{22} \frac{1}{2^1} + \dots + b_0 \frac{1}{2^{23}}; \quad (15.1)$$

- zmiennoprecinkowa **double** - 64-bitowa (S - 1 bit, E - 11 bitów, M - 52 bity):

$$x = (-1)^S \cdot M \cdot 2^{(E-1023)}, \quad S \in \{0, 1\}, E = 0, 1, \dots, 2047, M = 1 + b_{51} \frac{1}{2^1} + \dots + b_0 \frac{1}{2^{52}}; \quad (15.2)$$

- stałoprzecinkowa **QM.N** (M - liczba bitów przed przecinkiem, N - liczba bitów po przecinku, łącznie $M+N$):

$$\text{reprezentacja:} \quad \text{QM.N} = b_{M-1} \dots b_1 b_0 . b_{-1} b_{-2} \dots b_{-N}, \quad (15.3)$$

$$\text{wartość:} \quad x = b_{M-1} \cdot \left(-2^{(M-1)} \right) + \sum_{k=0}^{M+N-2} b_{k-N} \cdot 2^{(k-N)}, \quad (15.4)$$

$$\text{zakres:} \quad -2^{(M-1)} \leq x \leq 2^{(M-1)} - \frac{1}{2^N}, \quad (15.5)$$

$$\text{kwant:} \quad \Delta x = \frac{1}{2^N} (\text{LSB}), \quad (15.6)$$

$$\text{błąd:} \quad -\frac{1}{2^{(N-1)}} \leq \epsilon_x \leq \frac{1}{2^{(N-1)}}. \quad (15.7)$$

$$(15.8)$$

15.2 Liczby zmiennoprecinkowe

We wzorach (15.1) i (15.2) E oznacza wykładnik liczby 2, natomiast M - mantysę (część ułamkową liczby). Na rysunku 15.1 przedstawiono graficznie definicję zmiennoprecinkowej reprezentacji liczb typu *float* oraz *double*. Należy podkreślić, że wykładnik dobiera się tak, aby po przesunięciu przecinka liczba ułamkowa zawsze miała liczbę 1 przed przecinkiem, czyli była z zakresu $[1.0, 2.0)$ (jeśli wielkość liczby to umożliwia) — w takim przypadku w mantysie zapisywana jest tylko część ułamkowa bez wiodącej (poprzedzającej) jedynki, która jest domyślna. Domyślna jedynka zwiększa liczbę bitów reprezentacji i poprawia precyzję.

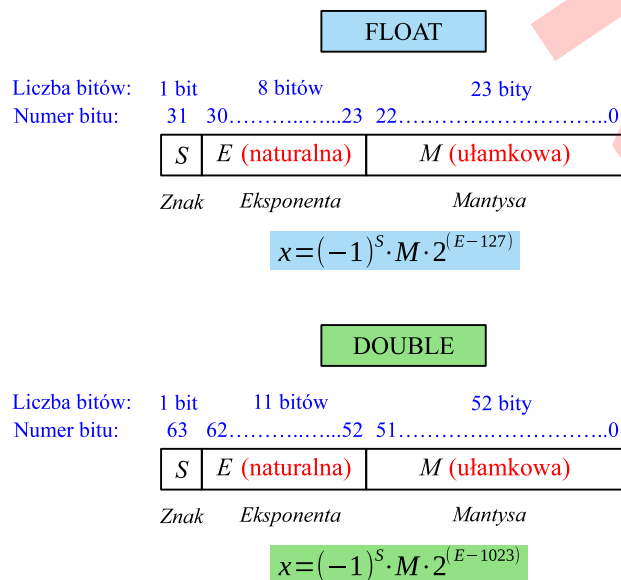
Format zmiennoprecinkowy daje duże poszerzenie zakresu reprezentacji liczb, ale kosztem ich precyzji. Porównując ze sobą 32-bitowe liczby zmiennoprecinkowe

float oraz stałoprzecinkowe *long*, zakres liczbowy w formacie zmiennoprzecinkowym jest $2^8 = 256$ razy większy, ale dokładność mantysy w formacie stałoprzecinkowym jest o 8 bitów lepsza. W obliczeniach zmiennoprzecinkowych nie ma ryzyka przepełnienia, ale istnieje olbrzymie niebezpieczeństwo utraty precyzji, w sytuacji kiedy wykonujemy operację arytmetyczną na dwóch liczbach znacznie różniących się wykładnikiem (przed wykonaniem operacji na liczbach zapisanych w reprezentacji *float* ustawiamy wykładnik liczby mniejszej tak, aby był on równy wykładnikowi liczby większej, w wyniku czego przecinek w części ułamkowej liczby mniejszej przesuwamy się w lewo, a bity przesuwają się w prawo i częściowo lub całkowicie nie mieszczą się już w reprezentacji bitowej mantysy, czyli są tracone). Sytuacja taka jest przedstawiona na rysunku 15.2 i porównana z dodawaniem liczb stałoprzecinkowych formatu *long*. Poniżej jako przykład jest podana konkretna, 32-bitowa liczba zmiennoprzecinkowa :

$$1\ 01111000\ 010\ 0000\ 0000\ 0000\ 0000\ 0000b = (-1)^1 \cdot 2^{(120-127)} \cdot \left(1 + \frac{1}{4}\right) =$$

$$= -1.25 \cdot 2^{-7} = -0.9765625 \cdot 10^{-2} \quad (15.9)$$

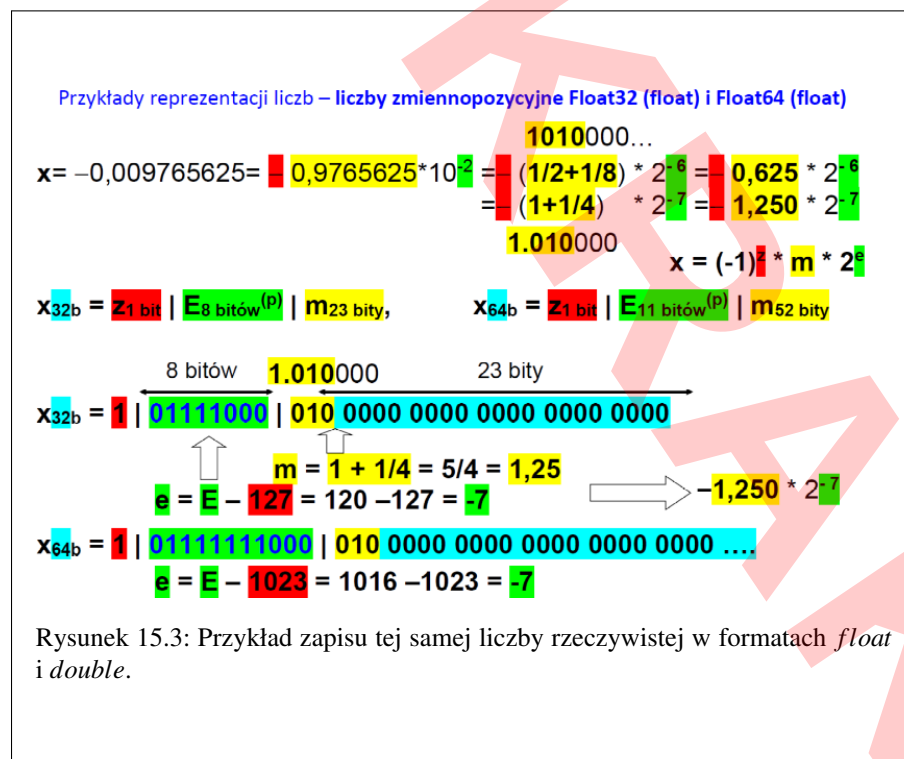
Na rysunku 15.3 jest przedstawiony przykład zapisu tej samej liczby $-0.009765625 = -0.9765625 \cdot 10^{-2}$ w formacie *float* oraz *double*. Na rysunkach 15.4 i 15.5 pokazano szczególne przypadki (wyjątki) zapisu liczb typu zmiennoprzecinkowego, na które trzeba bardzo uważać. Na koniec, na rysunku 15.6 zaprezentowano więcej przykładów konwersji liczb rzeczywistych na typ *float*.



Rysunek 15.1: Definicja liczb zmiennoprzecinkowych typu *float* oraz *double*.

Poprawny wynik		$2^{(-30)}$
$0.5 + 0.00000000931322574 = 0.50000000931322574$		
Zmiennoprzecinkowo		
		<i>Eksponenta</i> <i>Mantysa</i>
0.5	$=$	$00111111000000000000000000000000b$
$+ 0.00000000931322574$	$=$	$00110000100000000000000000000000b$
$= 0.50000000931322574$	$=$	$00111111000000000000000000000000b$
Stałoprzecinkowo		
0.5	$=$	$01000000000000000000000000000000b$
$+ 0.00000000931322574$	$=$	$00000000000000000000000000000000b$
$= 0.50000000931322574$	$=$	$01000000000000000000000000000000b$

Rysunek 15.2: Ilustracja utraty precyzji w liczbach typu *float*, w przeciwieństwie do liczb typu *fixed – point*.



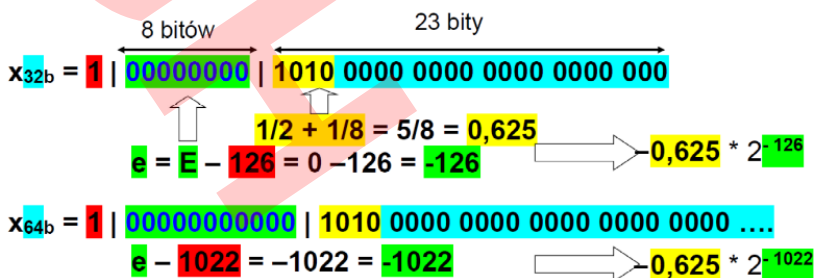
Przykłady reprezentacji liczb – liczby zmiennopozycyjne Float32 (float)

Ale trzeba być czujnym !!!

Jeśli E = same zera, to bity m nie są przesuwane w o 1bit lewo.

Tzn. nie są normalizowane.

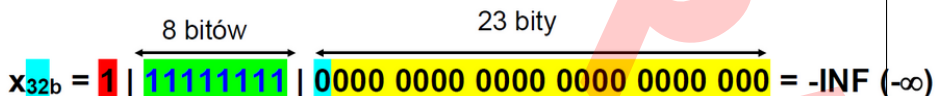
Dodatkowo wówczas: $e = E - 126$, $e = E - 1022$. Przykładowo:



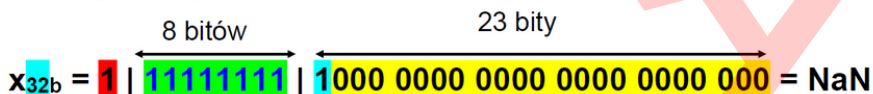
Rysunek 15.4: Przypadek liczby *float* z zerowym wykładnikiem E .

Jeśli E = same jedynki, to:

- a najstarszy bit $m_{MSB}=0$, to $\pm \infty$ (Infinity):


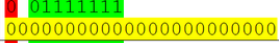
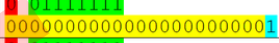
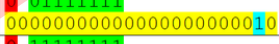
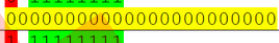

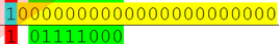
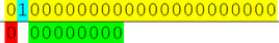
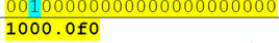


- a najstarszy bit $m_{MSB}=1$, to NaN (Not A Number):



Rysunek 15.5: Przypadek liczby *float* z wykładnikiem E zawierającym same bity 1.

Przykłady reprezentacji liczb Float32 w Julii

Komenda	Wynik	Interpretacja
<code>eps(Float32)</code>	1.1920929f-7	Dokładność dla Float32 (2^{-23})
<code>bits(eps(Float32))</code>		$(-1)^0 * 2^{(127-104)} * (1+0) = 2^{-23}$
<code>bits(1.+eps(Float32)/2)</code>		$(-1)^0 * 2^{(127-127)} * (1+0) = 2^0(1)=1$
<code>bits(1.+eps(Float32))</code>		$(-1)^0 * 2^{(127-127)} * (1+2^{-23}) = 1+2^{-23}$
<code>bits(1.+2*eps(Float32))</code>		$(-1)^0 * 2^{(127-127)} * (1+2^{-22}) = 1+2*2^{-23}$
<code>bits(Float32(Inf))</code>		„Nieskończoność” Inf
<code>bits(Float32(-Inf))</code>		„Minus Nieskończoność”, -Inf
<code>bits(Float32(NaN))</code>		„Not a Number”, NaN
<code>bits(Float32(-0.009765625))</code>		$(-1)^1 * (1+1/4) * 2^{-7}$ -unormowana
<code>bits(bits(Float32(2.)) ^... (-126-3))</code>		$(-1)^0 * 2^{-(126+3)}$ – NIE unormowana
<code>Float32((10.)^3)+... Float32((10.)^(3-8))</code>	1000.0f0	Druga liczba „zniknęła”

Rysunek 15.6: Przykłady reprezentacji bitowej różnych liczb zmiennoprzecinkowych.

Problem 15.1 (* Romeo i Julia). Załóż sobie konto na stronie: <https://juliahub.com/ui/Home>, np. używając konta Google albo pocztowego, oraz pobaw się różnymi reprezentacjami liczb w Julii. Na początek sprawdź proste sekwencje instrukcji, podane na rysunku 15.6. Potem zacznij je modyfikować. Korzystaj z opisu (# komentarz) i fragmentów programu 15.1.

Listing 15.1: Typy danych i proste komendy w Julii

```
# Typy danych:
# Int8, Int16, Int32, Int64, Int128
# UInt8, UInt16, UInt32, UInt64, UInt128
# BigInt
# Float16, Float32, Float64
# BigFloat
# Inf, -Inf, NaN

# Deklaracja i inicjalizacja
A = Int64(1234); F = Float16(1.125); B = Float64(A);

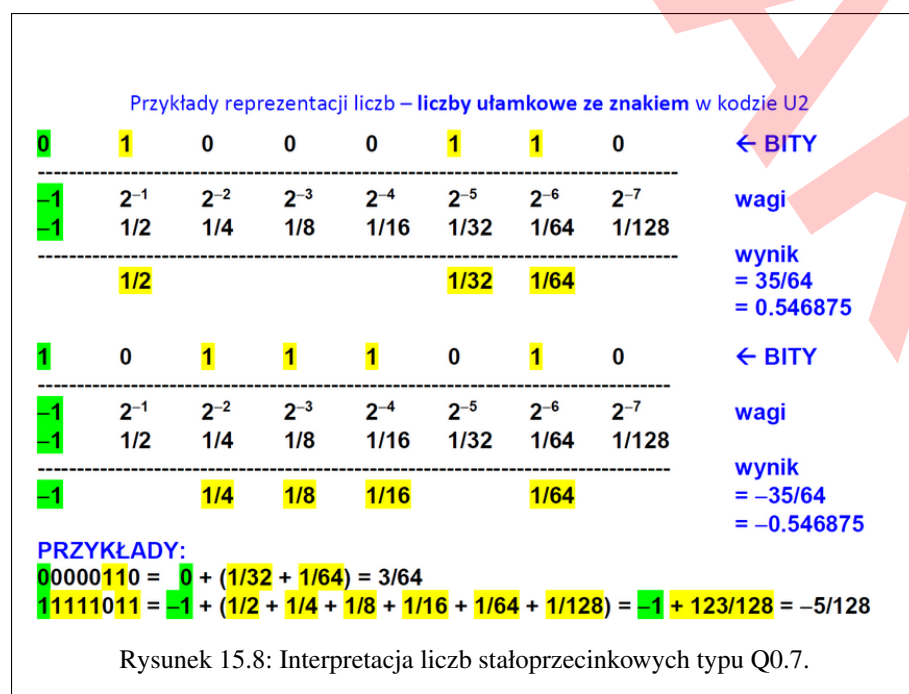
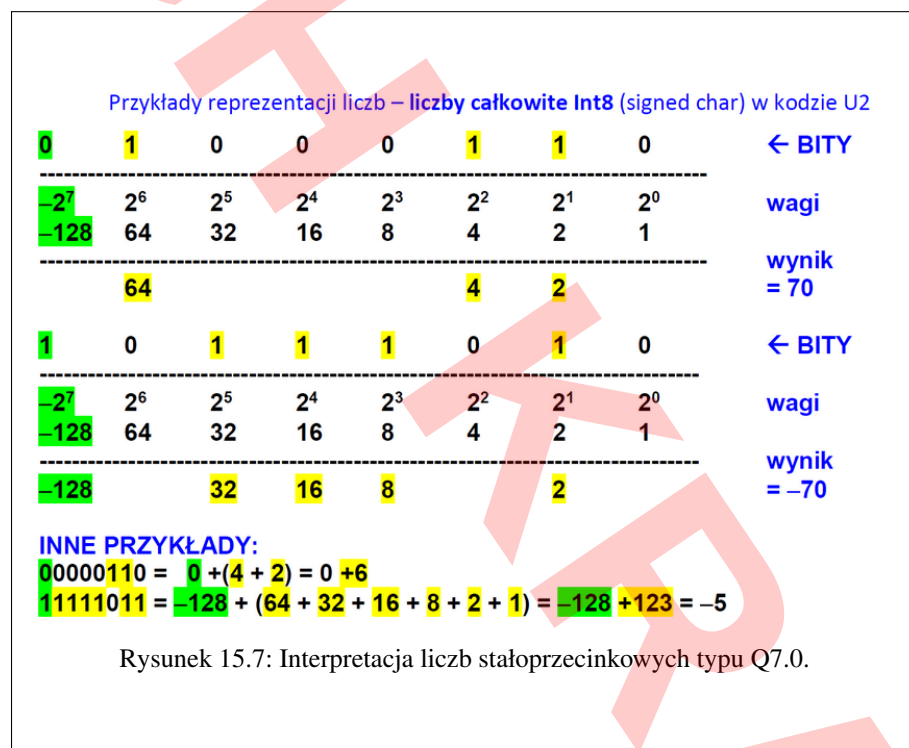
# Proste funkcje na zmiennych
tol = eps(Float16), eps(Float32), eps(Float64), eps(BigFloat)
Float16(1.) + eps(Float64);
a = Float16(1.25); b=a+eps(Float16);
bits(b);
```

15.3 Liczby stałoprzecinkowe

Najstarszy bit w reprezentacji QM.N to bit znaku: dzięki niemu otrzymujemy liczby ujemne i dodatnie kodowane w notacji U2 (uzupełnień do dwóch). Dla przykładu zinterpretujmy bity następującej liczby w zapisie Q1.31:

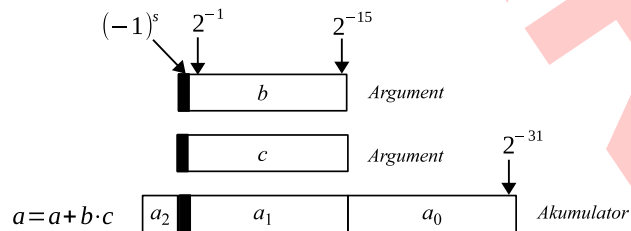
$$010100000000000000000000b = 0 \cdot (-2^0) + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + 1 \cdot \frac{1}{8} + \dots = 0.625 \quad (15.10)$$

Na rysunkach 15.7 oraz 15.8 w sposób poglądowy przedstawiono znaczenie poszczególnych bitów dla liczb 8 bitowych interpretowanych jako typ $Q7.0/int8/char/$ oraz jako typ $Q0.7/fract8$.



Kod QM.0 jest najczęściej stosowany w tanich mikrokontrolerach i procesorach ogólnego przeznaczenia, natomiast kod Q0.N — w dedykowanych procesorach sygnałowych (DSP *Digital Signal Processors*). Zamińmy się tym drugim. Co prawda ma on zakres ograniczony do $[-1, 1)$, ale posiada następujące zalety:

- operacja mnożenia $a = b \cdot c$ nie powoduje przepełnienia zakresu $[-1, 1)$, gdyż wynik mnożenia dowolnych dwóch liczb z tego zakresu należy również do tego zakresu; ponieważ w wyniku mnożenia liczby ułamkowe stają się "jedyne" mniejsze (np. $\frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16}$, $2^{-8} \cdot 2^{-8} = 2^{-16}$) w celu zachowania precyzji wynikowa liczba jest zapisywana w notacji Q0.(N+N) - do tego służą akumulatory wyników o podwojonej długości/precyzji, których zawartości można przesuwac (patrz rys. 15.9);
- operacja dodawania $a = b + c$ może dać liczbę dwukrotnie większą, czyli w formacie Q1.M, np. $(-1) + (-1) = -2$; w celu uniknięcia przepełnienia akumulatory mają z przodu dodatkowe bity "przechwytyjące" przepełnienie zakresu - rejestr a_2 na rys. 15.9;
- operacja mnożenia i akumulacji (MAC - *Multiply and Accumulate*) $a = a + b \cdot c$ (dodawania wyniku mnożenia do poprzedniej zawartości akumulatora) może dać liczbę w formacie Q1.(N+N), np. $(-1) + (1)(-1) = -2$ oraz $0 + \frac{1}{2^8} \cdot \frac{1}{2^8} = \frac{1}{2^{16}}$ (np. 24-bitowy procesor DSP56300 ma akumulatory o długości 56 bitów: 8+24+24) - 8 bitów z przodu na ewentualne przepełnienie i 24 bity z tyłu, zabezpieczające przed utratą precyzji - patrz rys. 15.14).



Rysunek 15.9: Rola akumulatora w operacjach na liczbach stałoprzecinkowych (dodatkowe bity z przodu i z tyłu).

15.4 Typ stałoprzecinkowy 16-to bitowy

Powszechnie używany i wystarczający w większości zastosowań jest typ stałoprzecinkowy implementowany na słowie typu `int` (eger) 16-to bitowym. Tabela 15.1 przedstawia dokładność reprezentacji typami od Q0.15 (powszechnie stosowany w procesorach DSP) do Q15.0 (zwykły `int16` w językach programowania).

Typ QM.N jest tylko specjalną interpretacją słowa bitowego typu `int`. Konwersja między typami jest prostym skalowaniem, co pokazuje tabela 15.2: inna reprezentacja jest otrzymywana poprzez podzielenie reprezentacji Q15.0 przez współczynnik skalujący.

Problem 15.2 ((L)* Testowanie formatu liczb QM.N). Zapoznaj się z programem 15.2. Sprawdź jego poprawność dla różnych formatów liczb QM.N. Napisz swój program.

Tabela 15.1: Dokładność reprezentacji liczb typu QM.N

Typ	Najmniejsza ujemna liczba	Największa dodatnia liczba	Dokładność (precyzja)
<i>QM.N</i>	-2^M	$2^M - \frac{1}{2^N}$	$\frac{1}{2^N}$
Q0.15	-1	0.999969482421875	0.00003051757813
Q1.14	-2	1.99993896484375	0.00006103515625
Q2.13	-4	3.9998779296875	0.00012207031250
Q3.12	-8	7.999755859375	0.00024414062500
Q4.11	-16	15.99951171875	0.00048828125000
Q5.10	-32	31.9990234375	0.00097656250000
Q6.9	-64	63.998046875	0.00195312500000
Q7.8	-128	127.99609375	0.00390625000000
Q8.7	-256	255.9921875	0.00781250000000
Q9.6	-512	511.984375	0.01562500000000
Q10.5	-1024	1023.96875	0.03125000000000
Q11.4	-2048	2047.9375	0.06250000000000
Q12.3	-4096	4095.875	0.12500000000000
Q13.2	-8192	8191.75	0.25000000000000
Q14.1	-16384	16383.5	0.50000000000000
Q15.0	-32768	32767	1.00000000000000

Tabela 15.2: Konwersja liczb reprezentacji typu QM.N poprzez skalowanie

Typ	Dzielnik (2^N)	Minimum HEX	Maksimum HEX
<i>QM.N</i>	2^N	8000h (-2^M)	7FFFh ($2^M - \frac{1}{2^N}$)
Q0.15	$2^{15} = 32768$	8000h (-1)	7FFFh (0.99)
Q1.14	$2^{14} = 16384$	8000h (-2)	7FFFh (1.99)
Q2.13	$2^{13} = 8182$	8000h (-4)	7FFFh (3.99)
Q3.12	$2^{12} = 4096$	8000h (-8)	7FFFh (7.99)
Q4.11	$2^{11} = 2048$	8000h (-16)	7FFFh (15.99)
Q5.10	$2^{10} = 1024$	8000h (-32)	7FFFh (31.99)
Q6.9	$2^9 = 512$	8000h (-64)	7FFFh (63.99)
Q7.8	$2^8 = 256$	8000h (-128)	7FFFh (127.99)
Q8.7	$2^7 = 128$	8000h (-256)	7FFFh (255.99)
Q9.6	$2^6 = 64$	8000h (-512)	7FFFh (511.99)
Q10.5	$2^5 = 32$	8000h (-1024)	7FFFh (1023.99)
Q11.4	$2^4 = 16$	8000h (-2048)	7FFFh (2047.99)
Q12.3	$2^3 = 8$	8000h (-4096)	7FFFh (4095.99)
Q13.2	$2^2 = 4$	8000h (-8192)	7FFFh (8191.99)
Q14.1	$2^1 = 2$	8000h (-16384)	7FFFh (16383.99)
Q15.0	$2^0 = 1$	8000h (-32768)	7FFFh (32767)

Listing 15.2: Program Matlaba do konwersji liczb stałoprzecinkowych QM.N do postaci binarnej/bitowej i z powrotem

```
% fractional2binary.m
a = 1234.57849; % your float point number
n = 16;         % number bits for integer part of your number
m = 20;         % number bits for fraction part of your number
% binary number
d2b = [ fix(rem(fix(a)*pow2(-(n-1):0),2)), fix(rem( rem(a,1)*pow2(1:m),2))], %
% the inverse transformation
b2d = d2b*pow2([n-1:-1:0-(1:m)].'),
```


15.5 Podstawowe operacje arytmetyczne w implementacji stałoprzecinkowej

W tabeli 15.3 scharakteryzowano wykonywanie podstawowych operacji arytmetycznych w arytmetyce stałoprzecinkowej QM.0 - liczby całkowite ze znakiem (U2). Czyli dla reprezentacji z dzielnikiem 1 w tabeli 15.2.

Tabela 15.3: Charakterystyka operacji arytmetycznych w implementacji stałoprzecinkowej typu QM.0 (najpierw jest wykonywana operacja w nawiasach (.), $x \gg M$ i $x \ll M$ oznacza przesunięcie M bitów w prawo/lewo)

Nazwa operacji	Wykonanie operacji	Uwagi
Dodawanie	$r = x + y$	zwykła arytmetyka U2, niebezpieczeństwo przepełnienia zakresu
Odejmowanie	$r = x - y$	zwykła arytmetyka U2, niebezpieczeństwo przepełnienia zakresu
Mnożenie	$r = (x * y) \gg M$	1) niebezpieczeństwo przepełnienia zakresu i utraty precyzji, 2) wynik pośredni dwukrotnie szerszy, 3) powielenie bitu znaku wyniku, jeśli liczby ze znakiem
Dzielenie	$(x \ll M) / y$	1) niebezpieczeństwo przepełnienia zakresu i utraty precyzji, 2) wynik pośredni dwukrotnie szerszy

15.6 Wybór reprezentacji liczb

Liczby zmiennoprzecinkowe typu float/double mają niezaprzeczalne zalety i wady.

Zalety liczb FP są następujące:

- duża dynamika zakresu,
- duża precyzja reprezentacji,
- łatwość programowania.

Wady liczb FP są następujące:

- Są nienaturalne dla niektórych platform/architektur, niemających FPU - *Floating Point Unit*:
 - stałoprzecinkowe procesory DSP taktowane najwyższymi częstotliwościami,
 - tanie, popularne mikrokontrolery 8/16-bitowe, np. 8051, V850,
 - układy FPGA (Field Programmable Gate Arrays), np. firm Xilinx i Altera.
- Obliczenia FP można emulować programowo, ale jest to powolne. Czas wykonania operacji arytmetycznych jest krytyczny w niektórych zastosowaniach, np. podczas dekodowania danych audio i wideo w czasie rzeczywistym, np. podczas strumieniowania multimediów w sieci komputerowej, oglądania cyfrowej telewizji naziemnej lub satelitarnej.
- Realizowane sprzętowo operacje zmiennoprzecinkowe zużywają więcej energii (prądu), generują więcej ciepła (problem jego odprowadzania) oraz zużywają więcej powierzchni krzemu w układzie procesora (powodują zwiększenie jego ceny!). A przecież urządzenia przenośne powinny być projektowane na minimalne zużycie baterii i jak najdłuższy czas działania.

Liczby stałoprzecinkowe typu *fixed-point*. W tym przypadku zalety i wady liczb zmiennoprzecinkowych zamieniają się miejscami, tzn. np. to co było wadą liczb zmiennoprzecinkowych staje się zaletą liczb stałoprzecinkowych, niestety za cenę utraty zalet tych pierwszych. Przykładowo, liczby stałoprzecinkowe mają następujące wady:

- małą dynamiką zakresu,
- małą precyzję reprezentacji kiedy liczba bitów jest ograniczona,
- trudność (czasochłonność) programowania.

Jednak znajdują szerokie zastosowanie z dwóch powodów:

- ponieważ oferują **małe zużycie energii (baterii) i długi czas działania procesorów**, są powszechnie stosowane w urządzeniach przenośnych: telefonach komórkowych, aparatach fotograficznych, układach nawigacji z odniornikami GPS, układach gier typu Game Boy,
- ponieważ **operują na natywnym formacie danych**, otrzymywanym bezpośrednio z urządzeń wejściowych, np. próbki dźwięku (kodery i dekodery audio MP3/AAC), kolor pikseli obrazu (konsole graficzne do gier X-Box/Play Station, odbiorniki TV), odczyt danych cyfrowych z czujników za pomocą przetworników typu analog-cyfra (stacje pogodowe).

W procesorach sygnałowych najczęściej stosuje się reprezentację stałoprzecinkową Q0.N. Natomiast w procesorach ogólnego przeznaczenia, np. mikrokontrolerach — reprezentację QM.0 (liczby całkowite ze znakiem w kodzie U2).

15.7 Przykłady implementacji stałoprzecinkowej algorytmów

Typowy schemat implementowania algorytmów stałoprzecinkowych, z powodu zagrożenia przepełnieniem, niedomiarem i utratą precyzji, zaczyna się od implementacji zmiennoprzecinkowej. Poprawnie działająca implementacja na typie *float/double* jest następnie zamieniana na typ QM.0 albo Q0.N i jest przeprowadzana analiza potencjalnych źródeł problemów. Na tym etapie można wspomagać się specjalizowanymi narzędziami, np. firma Mathworks sprzedaje pakiet oprogramowania *Fixed Point Designer* [1] do analizy *fixed-point*. Poniżej zapoznamy się z kilkoma wybranymi implementacjami stałoprzecinkowymi algorytmów.

FFT. Poniżej przedstawiono i porównano ze sobą implementację algorytmu FFT (Fast Fourier Transform) w wersji zmiennoprzecinkowej i stałoprzecinkowej.

Listing 15.3: Implementacja zmiennoprzecinkowa algorytmu FFT w Matlabie

```
function x = fi_m_radix2fft_algorithm1_6_2(x, w)
n = length(x); t = log2(n); % wartosci parametrow
x = fidemo.fi_bitreverse(x,n); % zmiana kolejnosci
for q = 1 : t % etapy
    L = 2^q; r = n/L; L2 = L/2; % stale w tym etapie
    for k = 0 : (r-1) % bloki
        for j = 0 : (L2-1) % motylki
            temp = w(L2-1+j+1) * x(k*L+j+L2+1); % korekta
            x(k*L+j+L2+1) = x(k*L+j+1) - temp; % # motylek
            x(k*L+j+1) = x(k*L+j+1) + temp; % # motylek
        end
    end
end
```

Listing 15.4: Implementacja stałoprzecinkowa algorytmu FFT w Matlabie

```
function x = fi_m_radix2fft_withscaling(x, w)
n = length(x); t = log2(n);
x = fidemo.fi_bitreverse(x,n);
% Generate index variables as integer constants
% so they are not computed in the loop.
LL = int32(2.^(1:t)); rr = int32(n./LL); LL2 = int32(LL./2);
for q = 1 : t
    L = LL(q); r = rr(q); L2 = LL2(q);
    for k = 0 : (r-1)
        for j = 0 : (L2-1)
            temp = w(L2-1+j+1) * x(k*L+j+L2+1); % korekta
            x(k*L+j+L2+1) = bitsra( x(k*L+j+1) - temp, 1); % motylek
            x(k*L+j+1) = bitsra( x(k*L+j+1) + temp, 1); % motylek
        end
    end
end
```

CORDIC. Listing 15.5 prezentuje kod programu w języku C (<http://www.dcs.gla.ac.uk/~jhw/cordic/>), implementujący algorytm CORDIC (patrz wykład "Funkcje standardowe"), służący do generowania wartości funkcji $\sin(n \cdot \Delta x)$ i $\cos(n \cdot \Delta x)$, dla kolejnych wartości kąta ($n = 0, 1, 2, \dots$). Jest to wersja stałoprzecinkowa: obrót wektora $[a, b]$ o większy kąt jest realizowany poprzez serię obrotów o mniejsze kąty Δx_k , dla których wartość funkcji $\tan(\Delta x_k)$ jest równa $\frac{1}{2^n}$. Dzięki temu zamiast mnożyć liczbę a lub b przez $\tan(\Delta x_k)$, wystarczy ją jedynie przemieścić n bitów w prawo. W związku z tym wykonuje się jedynie proste operacje dodawania, odejmowania i przesuwania bitów.

Obecnie, poniżej, zacytujemy fragment ze strony <http://www.dcs.gla.ac.uk/~jhw/cordic/>. "Na początku program generuje zbiór nagłówkowy ze stałymi oraz kod obliczeniowy algorytmu dla różnych długości słów komputerowych (16 lub 32 bity), oraz zapisuje zbiór nagłówkowy na standardowe urządzenie wyjściowe. Kolejność jest następująca.

1. Na początku program powinien być uruchomiony na komputerze posiadającym *Floating-Point Unit* (FPU) - do generacji stałych jest używana standardowa biblioteka matematyczna.
2. Potem wynikowy skrypt `*.h` powinien być skompilowany na docelowy procesor/platformę (np. mikrokontroler) za pomocą cross-kompilatora.

Należy podać liczbę bitów `bitsize` używanego słowa komputerowego: spowoduje to przepapowanie wartości 1 na $2^{\text{bitsize}-2}$. Przykładowo dla reprezentacji 16-bitowej plik nagłówkowy będzie używał formatu stałoprzecinkowego Q2.14, który w tym przypadku zapewni największą precyzję. Użytkownik może zmienić współczynnik mnożący, występujący poniżej, na inną wartość, jeśli zamierza użyć innej reprezentacji stałoprzecinkowej QM.N. Zapewni to maksymalną dokładność (do przetestowania jako zadanie domowe).

Funkcja CORDIC spodziewa się wartości argumentu z przedziału $[-\frac{\pi}{2}, \frac{\pi}{2}]$. Dla innych wartości argumentów wyniki są identyczne. Dla przedziału $[\frac{\pi}{2}, \pi]$ wynik jest taki sam jak dla $[\frac{\pi}{2}, \pi - \frac{\pi}{2}]$, podobnie dla przedziału $[-\pi, -\frac{\pi}{2}]$. Każdy kąt może być przepapowany do zakresu $[-\frac{\pi}{2}, \frac{\pi}{2}]$ biorąc jego wartość modulo π . W listingu 15.5 założono, że ustawiono `bitsize=32` oraz zapisano wyjście z programu `gentable.c` do zbioru `cordic-32bit.h`.

Oczywistym jest, że 16-bitowy algorytm CORDIC może być zaprogramowany podobnie prosto, dając w wyniku `cordic-16bit.h`. W zaprezentowanym listingu wyniki z 32-bitowego algorytmu CORDIC są porównywane z wynikiem standardowych implementacji funkcji `sin()` i `cos()` z biblioteki `math.h`. Jak widać algorytm nie jest idealny (perfekcyjnie dokładny), ale jego celem jest implementacja obliczeń na tanim sprzęcie: bez sprzętowych mnożarek lub z ograniczoną pamięcią.

Problem 15.3 (Analiza i testowanie programu CORDIC).** Sprawdź oprogramowanie CORDIC, pobrane ze strony <http://www.dcs.gla.ac.uk/~jhw/cordic/>, dla liczb stałoprzecinkowych 32, 16 i 8 bitowych.

Listing 15.5: Kod w języku C algorytm CORDIC w implementacji stałoprzecinkowej

```
// Pobrane ze strony: http://www.dcs.gla.ac.uk/~jhw/cordic/

#include "cordic-32bit.h"
#include <math.h> // for testing only!

//Print out sin(x) vs fp CORDIC sin(x)
int main(int argc, char **argv)
{
```

```
double p;
int s,c;
int i;
for(i=0;i<50;i++)
{
    p = (i/50.0)*M_PI/2;
    // use 32 iterations
    cordic(p*MUL), &s, &c, 32);
    // these values should be nearly equal
    printf("%f : %f\n", s/MUL, sin(p));
}

// cordic-32bit.h

//Cordic in 32 bit signed fixed point math
//Function is valid for arguments in range -pi/2 -- pi/2
//for values pi/2--pi: value = half_pi- (theta-half_pi) and similarly for values -pi--pi
/2
//
// 1.0 = 1073741824
// 1/k = 0.6072529350088812561694
// pi = 3.1415926536897932384626
//Constants
#define cordic_1K 0x2DD3B6A
#define half_pi 0x6487ED51
#define MUL 1073741824.000000
#define CORDIC_NTAB 32
int cordic_ctab[]={0x3243F6A8, 0x1DAC6705, 0x0FADBAFC, 0x07F56EA6, 0x03FEAB76, 0x01FFD55B
,
    0x00FFFAAA, 0x007FFF55, 0x003FFFEA, 0x001FFFFD, 0x000FFFFF, 0x0007FFFF
,
    0x0003FFFF, 0x0001FFFF, 0x0000FFFF, 0x00007FFE, 0x00003FFE, 0x00001FFF
,
    0x00000FFE, 0x000007FF, 0x000003FE, 0x000001FF, 0x000000FF, 0x0000007F
,
    0x0000003F, 0x0000001F, 0x0000000F, 0x00000008, 0x00000004, 0x00000002
,
    0x00000001, 0x00000000 };

void cordic(int theta, int *s, int *c, int n)
{
    int k, d, tx, ty, tz;
    int x=cordic_1K,y=0,z=theta;
    n = (n>CORDIC_NTAB) ? CORDIC_NTAB : n;
    for (k=0; k<n; ++k)
    {
        d = z>>31;
        //get sign. for other architectures you might want to use the more portable version
        //d = z>=0 ? 0 : -1;
        tx = x - (((y>>k) ^ d) - d);
        ty = y + (((x>>k) ^ d) - d);
        tz = z - ((cordic_ctab[k] ^ d) - d);
        x = tx; y = ty; z = tz;
    }
    *c = x; *s = y;
}
```

Cyfrowa filtracja rekursywna danych jest przykładem zastosowania implementacji stałoprzecinkowej, gdzie problemy numeryczne mogą być bardzo poważne. Obliczenia kolejnej wartości wyjściowej filtra rekursywnego mają postać ($x(n)$ - liczby wejściowe, $y(n)$ - liczby wyjściowe):

$$y(n) = \sum_{k=0}^M b_k x(n-k) - \sum_{k=1}^N a_k y(n-k), \quad (15.11)$$

Od strony obliczeniowej źródłem problemów jest zależność wielkości wyjściowej $y(n)$ od poprzednich wartości wyjściowych: $y(n-1), \dots, y(n-N)$. Tego typu sprzężenie zwrotne może generować bardzo duże i bardzo małe wartości pośrednie. Rodzi to niebezpieczeństwo przepełnienia, niedomiaru i utraty precyzji. Z tego powodu dane wyjściowe mogą zmieniać się niestabilnie, a nawet rosnać w sposób nieograniczony.

Aby ograniczyć wpływ błędu numerycznego na wynik, stosuje się specjalne struktury obliczeniowe, kaskadę tzw. sekcji bikwadratowych (czyli układów prostszych z opóźnieniami dwuelentowymi: typu $y(n) = \sum_{k=0}^2 b_k x(n-k) - \sum_{k=1}^2 a_k y(n-k)$). Wykonywana jest przez nią ta sama operacja co w równaniu (15.11), ale stabilniej. Analiza stabilności poszczególnych sekcji bikwadratowej jest dobrze opisana i umożliwia jej pełną kontrolę [2] [3].

15.8 Procesory sygnałowe

Cyfrowe układy analizy i przetwarzania danych pomiarowych, zbieranych z różnych czujników (np. temperatury, ciśnienia, naprężenia, prędkości, ...; z mikrofonów, anten, ...; cały świat dookoła nas: zastosowania multimedialne, medyczne, wojskowe i dowolne inne, autonomiczny samochód, inteligentny dom, przemysł, itp. itd.) są zasympywowane liczbami z przetworników analogowo-cyfrowych. Liczby te można interpretować jako liczby całkowite ze znakiem QM.0 (np. `int8`, `int12`, `int16`, ...) albo jako liczby ułamkowe Q0.M (np. `frac8`, `frac12`, `frac16`, ...). W tym drugim przypadku otrzymujemy liczby z przedziału $[-1, 1)$. Podstawową operacją wykonywaną przez procesor sygnałowy jest akumulowana suma (iloczyn skalarny) dwóch zbiorów liczb (dwóch sygnałów) $\{x_n\}$ oraz $\{y_n\}$:

$$acc = \sum_{n=1}^N x_n \cdot y_n \quad (15.12)$$

Występują trzy typowe sytuacje:

1. oba zbiory pochodzą z przetworników A/C (dane pomiarowe) i są ze sobą **korelowane** w celu znalezienia występującego podobieństwa pomiędzy nimi (np. odebrany sygnał radarowy jest korelowany z sygnałem nadanym w celu wyznaczenia opóźnienia pomiędzy nimi - na tej podstawie oblicza się odległość obiektu od radaru),
2. jeden zbiór, np. x_n , pochodzi z przetwornika A/C (dane pomiarowe), a drugi y_n jest sztucznie generowany i też jest z zakresu $[-1, 1)$; są to:
 - współczynniki filtra cyfrowego $y_n = h_n$, dzięki którym z sygnału x_n są **usuwane (odfiltrowane)** niechciane częstotliwości,
 - wartości liczbowe wzorca częstotliwości o kształcie funkcji sinus/kosinus, np. $y_n = \cos(n \cdot \Delta y)$ albo $y_n = \sin(n \cdot \Delta y) + j \cos(n \cdot \Delta y)$, dzięki którym wynik operacji 15.12 informuje nas o **sile występowania konkretnej częstotliwości** w analizowanym sygnale (dla częstotliwości f_0 : $\Delta y = 2\pi \frac{f_0}{f_{pr}}$, gdzie f_{pr} jest częstotliwością próbkowania).

Równanie (15.12) jest w praktyce implementowane jako akumulowana suma:

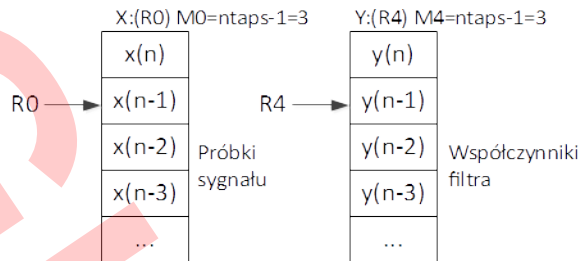
```
acc = 0;
acc = acc + x1 · y1;
acc = acc + x2 · y2;
acc = acc + x3 · y3;
...
```

Procesory sygnałowe są specjalnie projektowane do szybkiej realizacji sprzętowej akumulowanej sumy. Często mają one osobną pamięć P na program oraz dwa osobne bloki pamięci X i Y, z osobnymi magistralami adresowymi i danych, osobno dla danych x_n i y_n , które mogą pobierać równocześnie. Wykonują one sześć (6) operacji równocześnie dzięki obliczeniom potokowym:

- (2) - inkrementacja dwóch k -tych wskaźników do pamięci X i Y,
- (2) - pobranie kolejnych danych x_{k-1} i y_{k-1} ,
- (1) - wykonanie mnożenia wartości $x_{k-2} \cdot y_{k-2}$,
- (1) - dodanie wyniku mnożenia do aktualnej zawartości akumulatora acc .

Na rysunku 15.10 przedstawiono przykładowy program procesora Motorola/NXP DSP56300, napisany w asemblerze, służący do filtracji cyfrowej sygnału metodą średniej ważonej. W pamięci X jest sygnał x_n , a w pamięci Y – wagi filtra $y_n = h_n$. W kolejnych liniach są wykonywane następujące operacje:

- (1) wyzeruj akumulator CLR A, zapisz zawartość rejestru X0, z wartością z przetwornika A/C, do pamięci pod adres X: (R0) +, zwiększ rejestr adresowy R0 o 1, pobierz współczynnik filtra z pamięci Y: (R4) + do rejestru Y0, zwiększ rejestr adresowy R4 o 1,
- (1) powtórz (REP-eat) następną instrukcję NTAPS-1 razy,
- (3) pomnóż zawartość rejestru procesora X0 przez zawartość rejestru Y0 i dodaj do aktualnej zawartości akumulatora A (instrukcja MAC), pobierz do X0 kolejną próbkę sygnału z pamięci X spod adresu wskazywanego przez R0 - zwiększ R0 o 1, pobierz do Y0 kolejną próbkę sygnału z pamięci Y spod adresu wskazywanego przez R4 - zwiększ R4 o 1; powtarzaj to NTAPS-1 razy,
- wykonaj ostatnie mnożenie i zakumuluj wynik z zaokrągleniem (MACR), zmniejsz zawartość rejestru adresowego R0 o 1.



% Program dla procesora DSP56300 realizujący filtr FIR z rysunku powyżej

CLR	A	X0,X:(R0)+	Y:(R4)+,Y0	; zapisz x(n), pobierz współczynnik
REP	#NTAPS-1			; powtórz nast. instrukcję NTAPS-1 razy
MAC	X0,Y0,A	X:(R0)+,X0	Y:(R4)+,Y0	; splot
MACR	X0,Y0,A	(R0)-		; zaokrąglij wynik, przesun wskaźnik R0

Rysunek 15.10: Program filtracji cyfrowej metodą średniej ważonej, napisany w assemblerze procesora DSP56300.

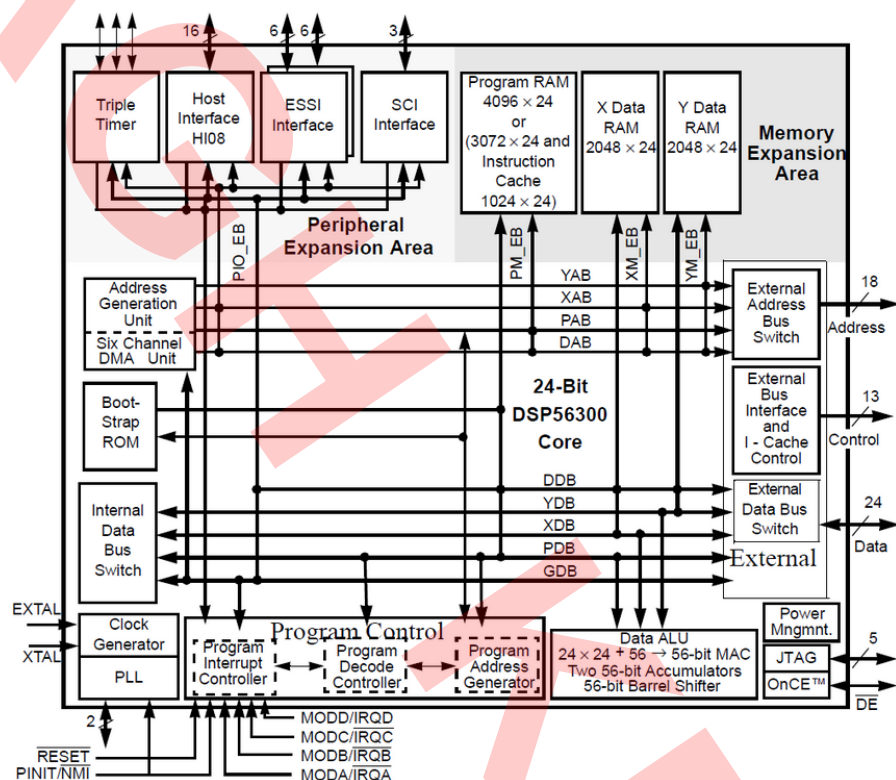
Nie wszystko może być jeszcze w tej chwili zrozumiałe, ale mam nadzieję, że za chwilę znaki zapytania znikną. Możesz wrócić do rysunku 15.10 po zapoznaniu się z opisem poniżej.

Na rysunku 15.11 jest przedstawiony [schemat blokowy 24-bitowego procesora stałoprzecinkowego DSP563000](#), z trzema blokami pamięci: P, X, Y. Oczywiście bez programu jest on tylko nikomu niepotrzebnym kawałkiem krzemu.

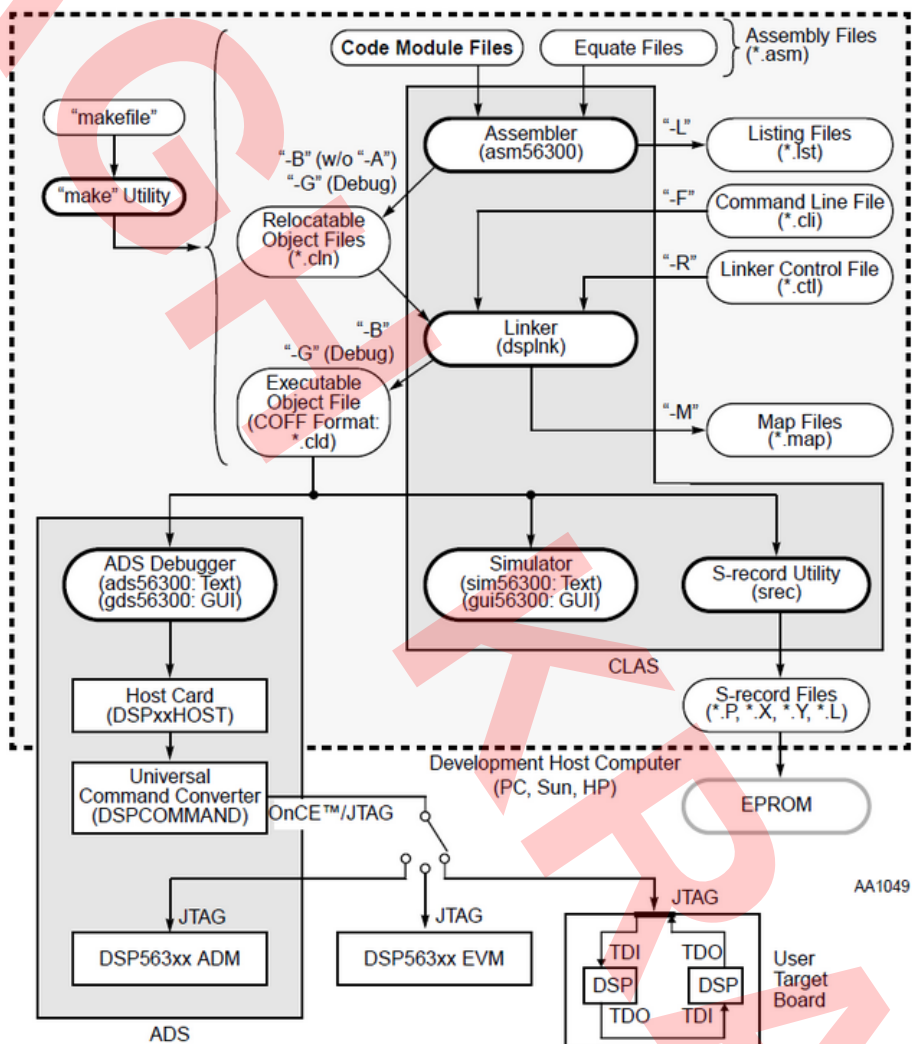
Jego [środowisko programowe](#) jest przedstawione na rysunku 15.12: [assembler](#), [linker](#), [symulator](#), możliwość podłączenia tzw. modułu uruchomieniowego *Evaluation Module* - szukaj w sieci "Symphony DSP56300 Studio Development Tools".

Na rysunku 15.13 jest przedstawiona [struktura logiczna \(programowalne rejestry\) procesora DSP56300](#): 4 rejestry 24-bitowe X1, X0, Y1, Y0 z możliwością połączenia w dwa rejestry 48-bitowe [X1X0], [Y1Y0], oraz dwa 56-bitowe akumulatory A, B. Dodatkowo 8 zestawów rejestrów adresowych o numerach x=0-7: ADRES=(Rx+Nx) modulo Mx (R-główny, N - inkrementacji/dekrementacji, M-modulo).

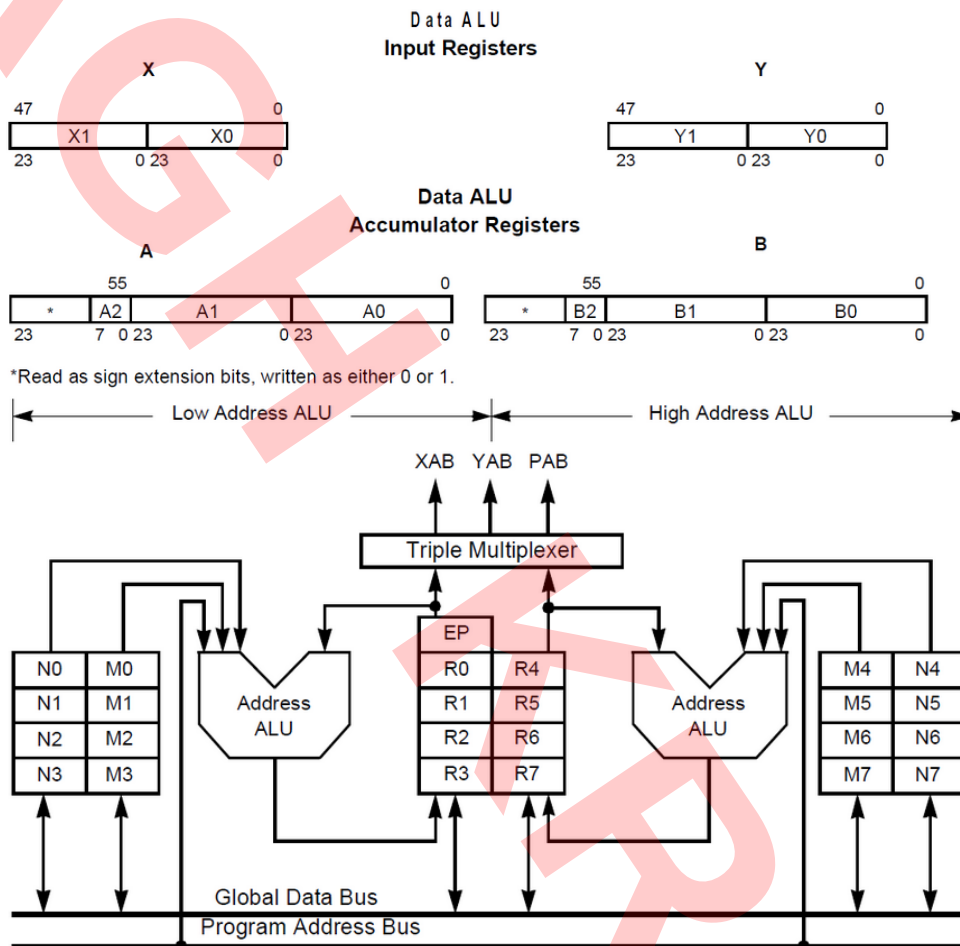
Na ostatnim rysunku 15.14 jest pokazana [jednostka arytmetyczno-logiczna \(ALU\) procesora](#): argumenty operacji mogą być 24-bitowe (rejestry: X0, X1, Y0, Y1) lub 48-bitowe (rejestry: [X1X0], [Y1Y0]), akumulatory A, B są 56-bitowe z zabezpieczeniem przed przepełnieniem i utratą precyzji.



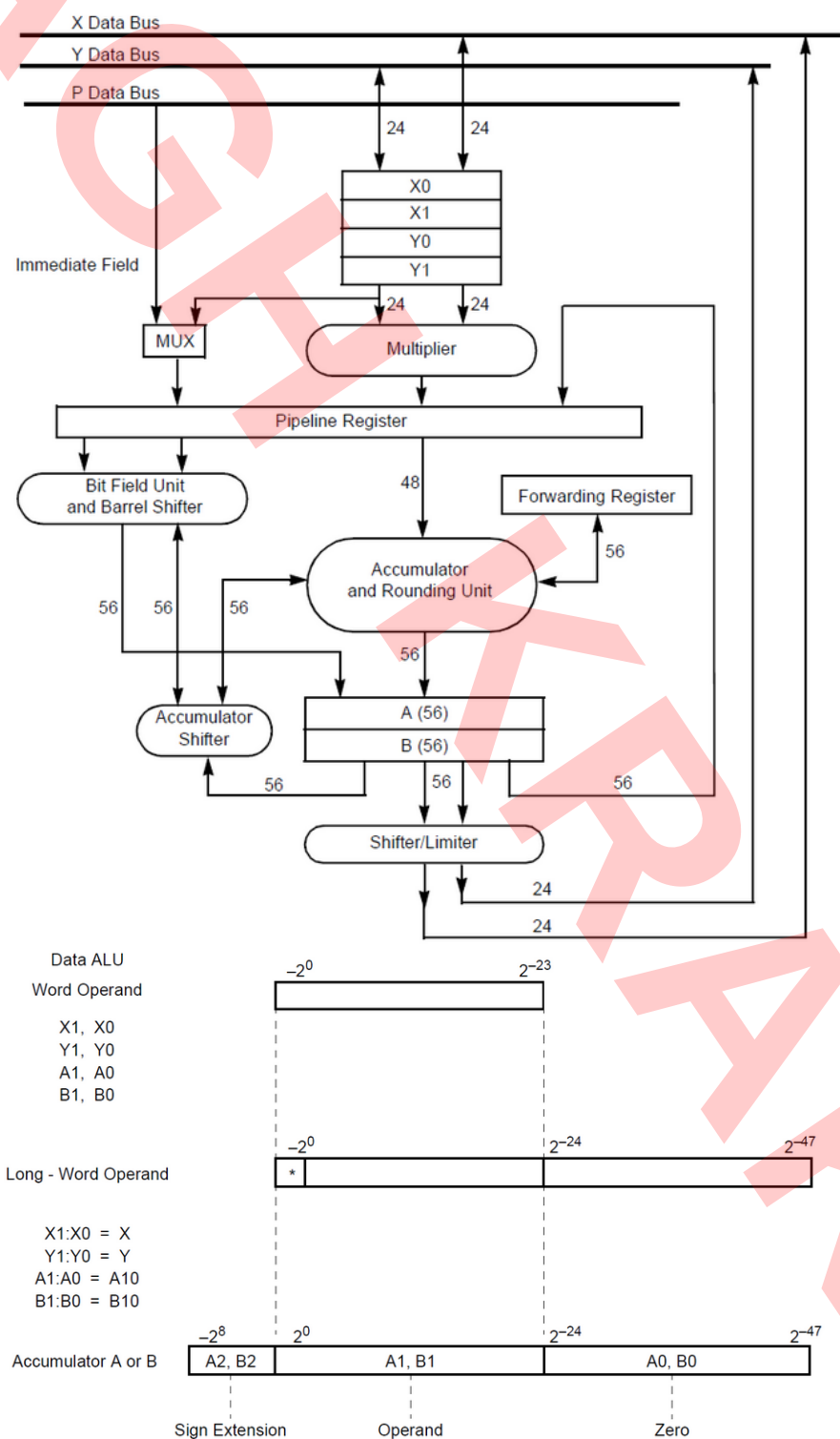
Rysunek 15.11: Schemat blokowy 24-bitowego procesora stałoprzecinkowego DSP563000.



Rysunek 15.12: Środowisko uruchomieniowe procesora DSP563000.



Rysunek 15.13: Rejestry wewnętrzne procesora SP56300: danych X, Y, A, B - podstawowe i akumulatory (góra) oraz adresowe R, N, M - główne, offsetowe, modulo (dół).



Rysunek 15.14: Jdnostka arytmetyczno-logiczna (ALU) procesora DSP563000 (góra) i rejestry danych - raz jeszcze (dół).

Problem 15.4 (* Zabawa assemblerem). Programowanie stałoprzecinkowe jest *niskopoziome*. Kod jest maksymalnie optymalizowany. Najczęściej na poziomie assemblera konkretnego procesora, a nie na poziomie języka C. Wejdź na stronę <https://godbolt.org/>. Napisz prosty program w języku C i zobacz jak on wygląda w assemblerze, tzn. jakie instrukcje procesora są używane.

15.9 Podsumowanie

Podsumujmy najważniejsze elementy tego wykładu, ostatniego w cyklu, w krótkich żołnierskich słowach.

- Część wykładnicza daje przewagę liczb typu *float* w sytuacji kiedy przetwarzane dane charakteryzują się dużą dynamiką/zmiennością.
- Wokół 0 format Q0.31 ma lepszą precyzję. Bity nie są tracone na niepotrzebną w tym przypadku część wykładniczą (*exponent*).
- Dla liczb większych i równych 1.0 następuje przepełnienie zakresu liczb Q0.31.
- Dla liczb małych dokładność reprezentacji Q0.31 spada.
- Liczby mniejsze od 2^{-31} leżą poza zakresem tego typu reprezentacji Q0.31.

Co z tego wynika?"

- typ stałoprzecinkowy można stosować w obliczeniach z **małą dynamiką** liczb,
- trzeba uważać na **przepełnienie** (*overflow*) i **niedomiar** (*underflow*) w trakcie obliczeń,
- błąd reprezentacji małych liczb może być duży,
- pociąg wjechał na końcową stację. Podróźni są proszeni o sprawdzenie czy nie pozostawili ... czegoś gdzieś ...

Literatura

1. Mathworks, *Fixed Point Designer*. Informacja on-line: <https://www.mathworks.com/products/fixed-point-designer.html>
2. W. Półchłopek, R. Rumian, P. Turcza, "Podstawy arytmetyki stałoprzecinkowej", w *Cyfrowe przetwarzanie sygnałów w telekomunikacji*, T. Zieliński, P. Korohoda, R. Rumian (red.), PWN, str. 394-429, 2014.
3. M. Christensen, F. J. Taylor, "Fixed-point-IIR-filter challenges", *EDN*, 2006, str. 1-9, on-line: <https://www.edn.com/fixed-point-iir-filter-challenges/>



AGH

ΑΓΗ

ΚΡΑΚΟΝ