

Realizacja projektu:

„Sumator dwóch liczb
zmiennoprzecinkowych o
pojedynczej precyzji,
w standardzie IEEE 754.”

Jan Bedliński

nr albumu 300480

Michał Mianowski

nr albumu 276171

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

1. Standard IEEE 754.

Standard **IEEE 754** definiuje dwa rodzaje liczb zmiennoprzecinkowych: **32-bitowe w pojedynczej precyzji** oraz

64-bitowe w podwójnej precyzji. Zajmować będziemy się tylko tą pierwszą wersją. Kod binarny liczby zmiennoprzecinkowej podzielony jest na trzy pola zawierające komponenty zapisu zmiennoprzecinkowego:

- a. Znak – 1 - bit – b_{31}
- b. Cecha/eksponent – 8 bitów – $b_{30} \dots b_{23}$
- c. Mantysa – 23 bity – $b_{22} \dots b_0$

W sumie : $1+8+23 = 32$ bity - $b_{31} \dots b_0$

a. Bit znaku

Najstarszy bit w zapisie liczby zwany jest bitem znaku. Stan 0 oznacza liczbę dodatnią, stan 1 liczbę ujemną. Aby zatem zmienić znak liczby zmiennoprzecinkowej na przeciwny, wystarczy dokonać negacji tego bitu.

b. Bity kodu cechy

Liczby zmiennoprzecinkowe **IEEE 754** zapisują cechę w kodzie z nadmiarem.

W pojedynczej precyzji cecha posiada 8 bitów, a nadmiar wynosi 127. Zatem w polu cechy można zapisać wartości od -127 (wszystkie bity $b_{30} \dots b_{23}$ wyzerowane) do 128 (wszystkie bity $b_{30} \dots b_{23}$ ustawione na 1).

c. Bity ułamkowe mantysy

W pojedynczej precyzji mantysa posiada 23 bity. W podwójnej 52 bity. Wzrost liczby bitów mantys liczb zmiennoprzecinkowych wpływa na ich precyzję, czyli *dokładność odwzorowywania liczb rzeczywistych*.

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

Mantysy są zapisywane w stałoprzecinkowym kodzie U1. Ponieważ mantysa jest prawie zawsze znormalizowana (z wyjątkiem wartości zdenormalizowanej, która jest przypadkiem szczególnym liczby zmiennoprzecinkowej IEEE 754), to jej wartość liczbową zawiera się pomiędzy 1 a 2. Wynika stąd, iż pierwszy bit całkowity mantysy zawsze wynosi 1. – ten wniosek wykorzystamy podczas opisu algorytmu dodawania.

2. Dodawanie liczb zmiennoprzecinkowych w standardzie IEEE 754.

Algorytm dodawania dwóch liczb zmiennoprzecinkowych, polega na kilku operacjach wykonanych w odpowiedniej kolejności:

1. Wyznaczenie z obu liczb: znaku, cechy i mantysy.
2. Normalizacja mantysy, w praktyce wiąże się to z dołączeniem **przed** najstarszy bit mantysy, **bit ‘1’**. Znormalizowana mantysa ma teraz 24 bity długości.
3. Wyrównanie cech liczb zmiennoprzecinkowych. Innymi słowy, jeżeli istnieje niezerowa różnica między eksponentami, należy wykonać przesunięcie mantysy w prawo **x** razy, gdzie **x** jest różnicą (liczbą naturalną w systemie dziesiętnym) między eksponentami.
4. Wykonujemy dodawanie, dwóch znormalizowanych mantys o równych (już) eksponentach.
5. W przypadku dodawania, może zdarzyć się, że wynik będzie o 1 bit dłuższy (25 bitowy), wtedy należy **zwiększyć o 1, wartość dziesiętną wyrównanej cechy**. Dodatkowo jeżeli długość się zwiększyła, (przypadek wyżej - 25 bitowa mantysa) najmłodszy bit mantysy jest tracony ze względu na fakt, że znormalizowana mantysa może mieć tylko 24 bity długości.
 - **Znak wynikowej liczby(1 bit)** jest **gotowy** od samego początku, gdyż jest to sumator, który wykonuje dodawanie dwóch liczb o tych samych znakach.
 - **Wynikowa cecha(8 bit-ów)** jest już również **gotowa**.
 - Mantysa po zsumowaniu posiada natomiast 24 bity długości, co jest nie dopuszczalne. Wynik sumatora miałby wtedy **33 bity!!!** co dla sumowania w kodzie np. U1 jest dopuszczalne, ale w standardzie IEEE 754 już nie.

Wynik dodawania znormalizowanych mantys jest również mantysą znormalizowaną co znaczy, że ma sztucznie wstawioną jedynekę przed najstarszy

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

bit. Dla przypomnienia jest to bit , oddzielający część ułamkową (czyli mantysę) od całości. Zatem:

6. Pozbycie się najstarszego bita(bita normalizującego), z wektora bitów mantysy - po operacji **mantysa(23 bity)** jest **gotowa**.
7. Połączenie wszystkich 3 elementów – wynik dodawania jest zawsze 32 bitowy.

3. Przykłady dodawania dwóch liczb w standardzie IEEE 754 - analitycznie.

Prześledźmy algorytm postępowania na przykładach.

Suma liczb X i Y przedstawionych w systemie IEEE 754:

Przykład pierwszy:

X = 0100 0010 0000 1111 0000 0000 0000 0000

Y = 0100 0001 1010 0100 0000 0000 0000 0000

Najpierw konwertujemy te liczby na postać decymalną, jednak z zachowaniem pewnych norm.

X = 0 | 100 0010 0 | 000 1111 0000 0000 0000 0000

S = 0 -> liczba X jest liczbą dodatnią

exp: 100 0010 0₍₂₎ = 132₍₁₀₎ $e_x = 132 - 127 = 5$

Z powyższymi danymi przechodzimy do konwersji mantysy do wartości X. Musimy jednak najpierw wykonać proces zwany - **normalizacją mantysy**. Mantysa jest prawie zawsze znormalizowana, to znaczy, że jej wartość liczbową mieści się w przedziale (1,2). Wynika stąd, że pierwszy bit całkowity mantysy wynosi ‘1’.

X = m = 1, 000 1111 0000 0000 0000 0000 * 2⁵

Y = 0 | 100 0001 1 | 010 0100 0000 0000 0000 0000

S = 0 -> liczba dodatnia

exp: 100 0001 1₍₂₎ = 131₍₁₀₎ $e_y = 131 - 127 = 4$

Y = m = 1, 010 0100 0000 0000 0000 0000 * 2⁴

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

Jeżeli występuje niezerowa różnica pomiędzy wartościami eksponentów należy wyrównać także eksponent do wartości tego wyższego. Przesuwamy mantysę w prawo o różnicę wartości eksponentów. W tym procesie najmłodsze bity są tracone.

$$e_x > e_y$$

$$e_x - e_y = 5 - 4 = 1$$

$$Y = 0,10100100000000000000000 * 2^5$$

W następnym kroku sumujemy znormalizowane mantysy:

	<i>1111</i>	
X	1,0001111	$*2^5$
Y	+ 0.1010010	$*2^5$
<hr/>		
	1,1100001	$*2^5$

Wyznaczamy na nowo eksponent. W tym przykładzie pozostaje on taki jak w X – większy eksponent, czyli 10000100₍₂₎

Wynikowa mantysa to 23-bitowa liczba po przecinku. Jest to wynik sumowania pomijając najstarszy bit – 1, którą sami dopisaliśmy przed dodawaniem ze względu na potrzebę normalizacji mantysy.

Wynik:

$$0 \mid 10000100 \mid 11000010000000000000000$$

$$W = 0100010001100001000000000000000$$

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

Przykład drugi:

A = 0110 1011 1111 0011 1010 0000 1100 0011

B = 0110 1011 1000 1110 0101 1111 0001 1100

A = 0110 1011 1111 0011 1010 0000 1100 0011

S = 0 -> liczba dodatnia

exp: 110 1011 1₍₂₎ = 215₍₁₀₎

W opisywanym algorytmie można pominąć odejmowanie w tym miejscu liczby(**nadmiaru**) 127, a na koniec dodawanie jej.

m = 1, 111 0011 1010 0000 1100 0011

B = 0110 1011 1000 1110 0101 1111 0001 1100

S = 0 -> liczba dodatnia

exp: 110 1011 1₍₂₎ = 215₍₁₀₎

m = 1, 000 1110 0101 1111 0001 1100

$e_a - e_b = 0$ -> nie ma potrzeby wyrównywania cech liczb zmiennoprzecinkowych

Właściwe dodawanie:

$$\begin{array}{r} \phantom{*2^{215}} \\ \phantom{*2^{215}} \\ A \phantom{*2^{215}} \\ B \phantom{*2^{215}} \\ \hline \phantom{*2^{215}} \end{array}$$

Otrzymany wynik ma dwa bity przed przecinkiem (**suma 25bitowa**), a powinien być tylko jeden, należy więc zwiększyć **wartość dziesiętną** eksponentu o 1. W wyniku tego z mantysy zostaje utracony najmłodszy bit (**znormalizowana mantysa - 24bitowa**).

Exp : 216₍₁₀₎ = 1101 1000₍₂₎

m = 1,10000001111111111101111

Wynik:

W₂ = 0110 1100 0100 0000 1111 1111 1110 1111

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

Przykład Trzeci:

C = 0110 1011 1111 0011 1010 0000 1100 0011

D = 0110 1000 1000 1110 0101 1111 0001 1100

C = 0110 1011 1111 0011 1010 0000 1100 0011

S = 0 -> liczba dodatnia

exp: 110 1011 1₍₂₎ = 215₍₁₀₎

m = 1,11100111010000011000011

D = 0110 1000 1000 1110 0101 1111 0001 1100

S = 0 -> liczba dodatnia

exp: 110 1000 1₍₂₎ = 209₍₁₀₎

m = 1,00011100101111100011100

$e_c - e_d = 215 - 209 = 6 \neq 0$ -> wymagana jest normalizacja cechy w liczbie D, bo $e_c > e_d$

$m_D = 0,00000100011100101111100011100$

Znormalizowana mantysa musi mieć długość 24 bitów.

Najmłodsze bity są tracone.

Właściwe dodawanie:

$$\begin{array}{rcl} & 1 & 1 & 111 \\ C & 1,11100111010000011000011 & *2^{215} \\ D & + & 0,00000100011100101111100 *2^{215} \\ \hline & 1,11101011101101000111111 & *2^{215} \end{array}$$

Nie ma potrzeby wyrównywać eksponentu.

Wynik:

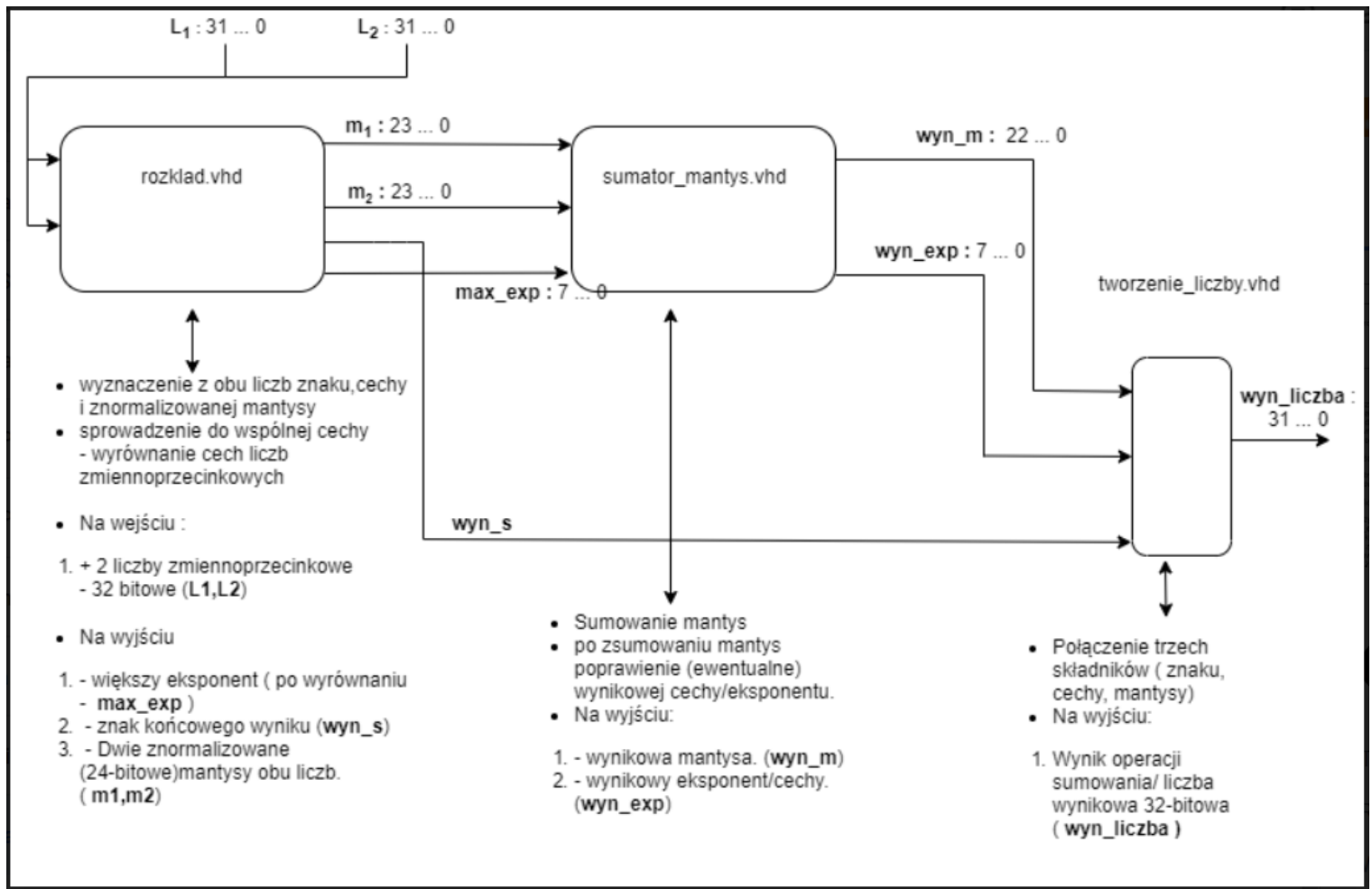
$W_3 = 0110 1011 1111 0101 1101 1010 0011 1111$

Powyższymi przykładami posłużymy się jako danymi testowymi w symulacjach.

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

4.Schemat blokowy projektu sumatora.

Schemat posiada podpisy konkretnych operacji, które są wykonywane w poszczególnych komponentach. **Porównamy go do schematu wygenerowanego przez Quartusa.**



Będziemy posiadać zatem 4 pliki *.vhd. 3 ostatnie pliki .vhd to widoczne na schemacie komponenty, natomiast tym pierwszym jest główna jednostka sterująca sygnałami wejściowymi, przejściowymi i wyjściowymi.

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

Opis wejść/wyjść

Nazwa Sygnału	Typ portu	Szerokość Szyny [bit]	Opis
liczba1	in	32	Pierwsza liczba będąca jednym ze składników sumy
liczba2	in	32	Druga liczba będąca drugim składnikiem sumy
mantys1	in/out	24	Znormalizowana mantysa pierwszej liczby
mantys2	in/out	24	Znormalizowana mantysa drugiej liczby
bit_znaku_wyniku	in/out	1	Najstarszy bit znaku wyniku sumowania
max_exp	in/out	8	Większy eksponent z obu liczb
mantysa_wyniku	in/out	23	Wynik sumowania mantys (zdenormalizowany)
exp_wyniku	in/out	8	Końcowa cecha/eksponent wyniku sumowania
wynik	out	32	Wynik sumowania dwóch liczb zmiennoprzecinkowych

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

5. Wykonanie projektu w Quartus Prime Lite.

Przedstawimy wszystkie komponenty napisane w języku opisu sprzętu – VHDL. Kod posiada komentarze, opisujące zasadę działania algorytmu.

1. Plik – **main_project.vhd** – deklaracja komponentów, sygnałów (w tym przejściowych) i ich mapowanie zgodnie ze schematem blokowym.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4
5  entity main_project is
6  port (
7      liczba1,liczba2 : in std_logic_vector(31 downto 0);
8      wynik : out std_logic_vector(31 downto 0)
9  );
10
11 end entity;
12
13 architecture main of main_project is
14     -- deklaracja component'ow
15
16
17     component rozklad
18     port (
19         -- Rozkład liczb na: znak, ceche/eksponent, mantyse
20         -- mantyse normalizujemy - wstawiamy bit 1-yuki przed najstarsza pozycje
21         -- należy znormalizowac mniejszy eksponent i ostatecznie poprawic mantyse liczby o mniejszym eksponencie
22         -- uwzględniamy różnice wartości eksponentów w mantysach i przesuwamy w prawo mantyse o mniejszym eksponencie
23         -- // shift_right(unsigned(...), ...);
24
25         -- @ wyznaczenie wynikowego znaku liczby (najstarszy bit) @
26
27         rozk_liczba1,rozk_liczba2 : in std_logic_vector(31 downto 0);
28         bit_znaku_wyniku : out std_logic;
29         mantys1, mantys2 : out std_logic_vector(23 downto 0);
30         max_exp : out std_logic_vector(7 downto 0)
31     );
32 end component;
33
34
35
```

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

```
36 component sumator_mantys
37 port (
38
39     -- sumujemy znormalizowane mantysy
40     -- pozbywamy sie juz najstarszego bitu z mantysy (ktorego dodalismy sztucznie)
41
42     mant_sum1, mant_sum2 : in std_logic_vector(23 downto 0);
43     maks_eksp : in std_logic_vector(7 downto 0);
44     exp_wyniku : out std_logic_vector(7 downto 0);
45     mantysa_wyniku : out std_logic_vector(22 downto 0)
46
47 );
48 end component;
49
50
51 component tworzenie_liczby
52 port(
53     wynikowa_mantysa : in std_logic_vector(22 downto 0);
54     wynikowy_exponent : in std_logic_vector(7 downto 0);
55     wynikowy_znak : in std_logic;
56     wynikowa_liczba : out std_logic_vector(31 downto 0)
57
58 );
59 end component;
60
61 signal tmp_mantys1,tmp_mantys2 : std_logic_vector(23 downto 0);
62 signal tmp_max_exponent, tmp_wynikowy_exp : std_logic_vector(7 downto 0);
63 signal tmp_wynikowa_mantysa : std_logic_vector(22 downto 0);
64 signal tmp_wynikowy_znak: std_logic;
65
66 begin
67
68
69 -- mapowanie portow componentow
70
71 rozklad_instancja : rozklad
72 port map(
73     rozk_liczba1 => liczba1,
74     rozk_liczba2 => liczba2,
75     mantys1 => tmp_mantys1,
76     mantys2 => tmp_mantys2,
77     max_exp => tmp_max_exponent,
78     bit_znaku_wyniku => tmp_wynikowy_znak
79 );
80
81
82 suma_instancja : sumator_mantys
83 port map(
84     mant_sum1 => tmp_mantys1,
85     mant_sum2 => tmp_mantys2,
86     maks_eksp => tmp_max_exponent,
87
88     exp_wyniku => tmp_wynikowy_exp,
89     mantysa_wyniku => tmp_wynikowa_mantysa
90 );
91
92
93 tworzenie_instancja : tworzenie_liczby
94 port map(
95     wynikowa_mantysa => tmp_wynikowa_mantysa,
96     wynikowy_exponent => tmp_wynikowy_exp,
97     wynikowy_znak => tmp_wynikowy_znak,
98     wynikowa_liczba => wynik
99 );
100
101 end main;
```

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

2. Plik – rozklad.vhd – opis funkcjonalny bloku.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  entity rozklad is
7  port (
8      rozk_liczba1, rozk_liczba2 : in std_logic_vector(31 downto 0);
9      bit_znaku_wyniku : out std_logic;
10     mantys1, mantys2 : out std_logic_vector(23 downto 0);
11     max_exp : out std_logic_vector(7 downto 0)
12 );
13
14 end entity;
15
16 architecture main of rozklad is
17
18 begin
19
20     process(rozk_liczba1, rozk_liczba2)
21
22         variable m1, m2 : std_logic_vector(23 downto 0);
23         variable e1, e2 : std_logic_vector(7 downto 0);
24         variable s1, s2 : std_logic;
25
26         variable tmp_e1, tmp_e2 : integer;
27         variable roznica_exp : integer;
28
29         begin
30             -- pierwsza liczba
31             s1 := rozk_liczba1(31);
32             e1 := rozk_liczba1(30 downto 23);
33             m1 := '1' & rozk_liczba1(22 downto 0);    -- normalizujemy mantyse wstawiajac bita 1 przed najstarszy bit
34
35             tmp_e1 := to_integer(unsigned(e1));
36
37             -- druga liczba
38             s2 := rozk_liczba2(31);
39             e2 := rozk_liczba2(30 downto 23);
40             m2 := '1' & rozk_liczba2(22 downto 0);    -- normalizujemy mantyse wstawiajac bita 1 przed najstarszy bit
41             tmp_e2 := to_integer(unsigned(e2));
42
43             if (s1='0' and s2='0') then
44                 bit_znaku_wyniku <= '0';
45                 -- wyznaczenie bitu znaku
46                 -- gdyby odejmowanie zostało zrobione tutaj nalezy zmodyfikowac if'a
47             else
48                 bit_znaku_wyniku <= '1';
49             end if;
50
51             if (tmp_e1 < tmp_e2) then
52
53                 roznica_exp := (tmp_e2 - tmp_e1);
54                 m1 := std_logic_vector(shift_right(unsigned(m1), roznica_exp));
55                 max_exp <= e2;
56                 -- przypisujemy juz maksymalna wartosc eksponentu
57             else
58
59                 roznica_exp := (tmp_e1 - tmp_e2);
60                 m2 := std_logic_vector(shift_right(unsigned(m2), roznica_exp));
61                 max_exp <= e1;
62                 -- przypisujemy juz maksymalna wartosc eksponentu
63             end if;
64
65             mantys1 <= m1;
66             mantys2 <= m2;
67
68         end process;
69
70     end main;
```

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

3. Plik – **sumator_mantys.vhd** – opis funkcjonalny bloku.

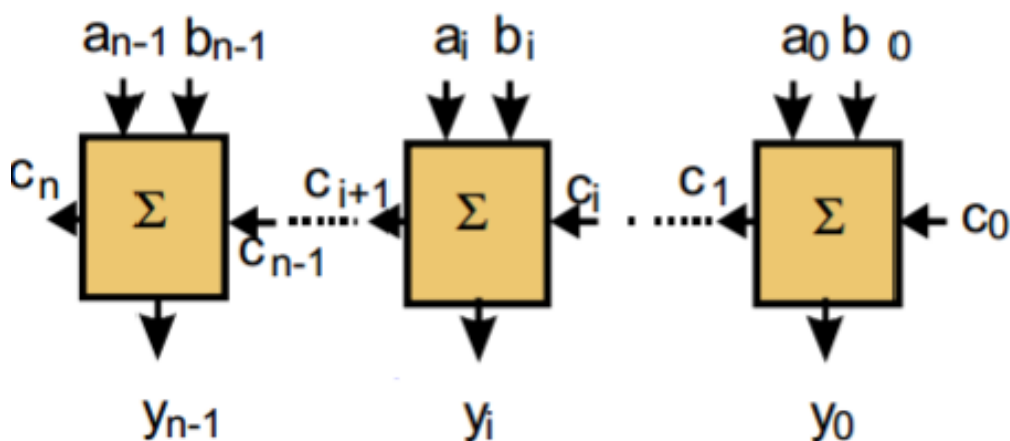
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  entity sumator_mantys is
7  port(
8      mant_sum1, mant_sum2 : in std_logic_vector(23 downto 0);
9      maks_eksp : in std_logic_vector(7 downto 0);
10     exp_wyniku : out std_logic_vector(7 downto 0);
11     mantysa_wyniku : out std_logic_vector(22 downto 0)
12 );
13 end entity;
14
15
16
17 architecture main of sumator_mantys is
18
19 begin
20
21     process(mant_sum1,mant_sum2) -- jako argument do processu [sensitive list] nie moze byc podany argument
22                                   -- ktory bedzie sie w procesie zmienial wynikiem bedzie infinity loop
23
24     variable wynik_moved, single_new_bit : std_logic;
25     variable tempA,tempB,tmp_suma: std_logic_vector(23 downto 0);
26     variable w_Sum : std_logic_vector(24 downto 0);
27
28     variable exp_int : integer;
29     variable tmp_exp : std_logic_vector(7 downto 0);
30
31 begin
32     tmp_exp := maks_eksp;
33
34     wynik_moved := '0'; -- przesuniecie
35     tempA := mant_sum1;
36     tempB := mant_sum2;
37
38     for i in 0 to 23 loop
39         single_new_bit := tempA(i) xor tempB(i) xor wynik_moved; -- przypisujemy bitowi wartosc xor z 3 bitow czyli defacto dodawania
40         w_Sum(i) := single_new_bit; -- przypisuje wynikowemu wektorowi i-tego bita
41
42         -- ogarniecie kiedy wynik_moved = 1 a kiedy wynik_moved = 0(przesuniecie)
43         wynik_moved := ( tempA(i) and tempB(i) ) or ( tempA(i) and wynik_moved ) or ( wynik_moved and tempB(i));
44     end loop;
45     w_Sum(24) := wynik_moved; -- dopisujemy na najstarszy bit wartosc przesuniecie
46
47     if wynik_moved = '0' then
48         tmp_suma := w_Sum(23 downto 0); -- jezeli ostatnie przesuniecie (wynik_moved) jest bitem '0' to jest on niepotrzebny
49         exp_wyniku <= tmp_exp; -- jezeli nie ma przesuniecie to exponent nowej liczby
50                                 -- jest wiekszym eksponentem z dwoch poprzednich liczb( rzad sie zgadza)
51     else
52         tmp_suma := w_Sum(24 downto 1); -- jezeli ostatnie przesuniecie (wynik_moved) jest bitem '1' to
53                                         -- nasza mantysa ma 25 bitow a nie 24 musimy zatem usunac najmłodszego bita
54                                         -- jezeli ostatnie przesuniecie jest bitem '1'
55                                         -- to znaczy ze musimy przesunac przecinek o jedno miejsce w lewo ,
56                                         -- zatem nasz eksponent wzrosnie o 1
57         exp_int := 1 + to_integer(unsigned(tmp_exp));
58         exp_wyniku <= std_logic_vector(to_unsigned(exp_int,8));
59     end if;
60
61     mantysa_wyniku <= tmp_suma(22 downto 0); -- mantysa wynikowa ma miec 23 bity , sa to bity za najstarszym bitem ( po przecinku)
62
63 end process;
64
65 end main;
```

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

- Przed przystąpieniem do przedstawienia ostatniego komponentu, warto opisać dodawanie mantys. Jest to zwykle dodawanie dwóch liczb w systemie dwójkowym. Wykorzystujemy zatem tutaj sumator dwóch liczb 24 bitowych.

```
35 wynik_moved := '0';           -- przesuniecie
36 tempA := mant_sum1;
37 tempB := mant_sum2;
38
39 for i in 0 to 23 loop
40     single_new_bit := tempA(i) xor tempB(i) xor wynik_moved;  -- przypisujemy bitowi wartosc xor z 3 bitow czyli defacto dodawania
41     w_Sum(i) := single_new_bit;                                -- przypisuje wynikowemu wektorowi i-tego bita
42
43     -- ogarniecie kiedy wynik_moved = 1 a kiedy wynik_moved = 0(przesuniecie)
44     wynik_moved := ( tempA(i) and tempB(i) ) or ( tempA(i) and wynik_moved ) or ( wynik_moved and tempB(i));
45
46 end loop;
47 w_Sum(24) := wynik_moved;  -- dopisujemy na najstarszy bit wartosc przesuniecia
49
```

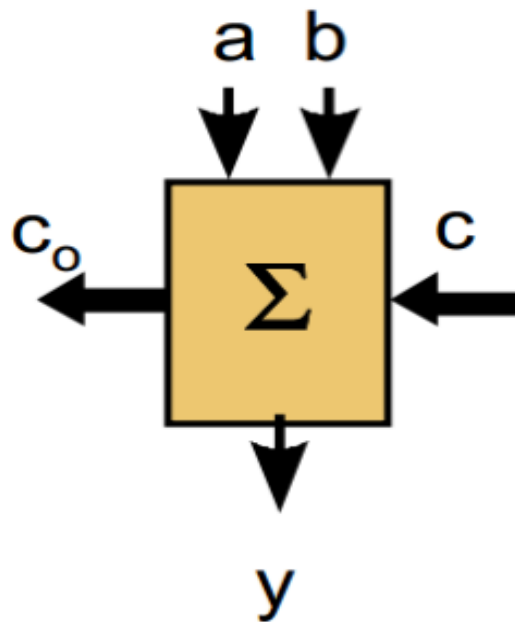
- W naszym projekcie, do sumowania mantys została zaimplementowana najprostsza wersja sumatora czyli: „*ripple carry adder*”



Dodawanie dwóch wielobitowych liczb dwójkowych może być realizowane szeregowo lub równolegle. W realizacji szeregowej kolejne pary bitów dodanej i dodajnika są sumowane wraz z odpowiednim przeniesieniem w szeregu cykli, dopóki dodawanie całego słowa nie zostanie zakończone.

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

- Przedstawimy sposób działania pojedynczego bloku i jego funkcje logiczne



a, b – bity które sumujemy

y – wynik sumowania

c – bit „przeniesienia”, które należy wziąć pod uwagę jeżeli jest bitem ‘1’

c_o – bit „przeniesienia”, który **może** powstać w wyniku operacji sumowania

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

Tabela prawdy				
a_i	b_i	c_i	y_i	c_{i+1}
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Jak wyznaczana jest aktualna wartość y_i oraz następna wartość przesunięcia c_{i+1} :

Z tabeli prawdy, łatwo zauważyć że dla nieparzystej ilości bitów **jedynek** (pierwsze 3 kolumny - $[a_i \ b_i \ c_i]$), wartość y_i jest równa '1', a dla parzystej ilości **jedynek** jest równa '0'. Operator pozwalający na uzyskanie takich wyników to operator XOR.

$y = c \text{ xor } a \text{ xor } b$	$y = c \oplus a \oplus b$
---------------------------------------	---------------------------

Wzór na wyznaczenie przesunięcia jest trochę bardziej skomplikowany i nie ma jednej wersji jak można by go zapisać. W naszym algorytmie jest zapisany w formie:

$$c_{i+1} = (a \text{ and } b) \text{ or } (a \text{ and } c_i) \text{ or } (b \text{ and } c_i)$$
$$c_{i+1} = (a * b) + (a * c_i) + (b * c_i)$$

Dany opis funkcjonalny bloku jest przedstawiony dla odpowiednich dwóch bitów, czyli w jednej iteracji.

Nasze mantysy mają 24bity długości zatem daną operację powtarzamy w pętli, aż każda para bitów zostanie zsumowana. (24 iteracje)

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

4. Plik – **tworzenie_liczby.vhd** – opis funkcjonalny bloku

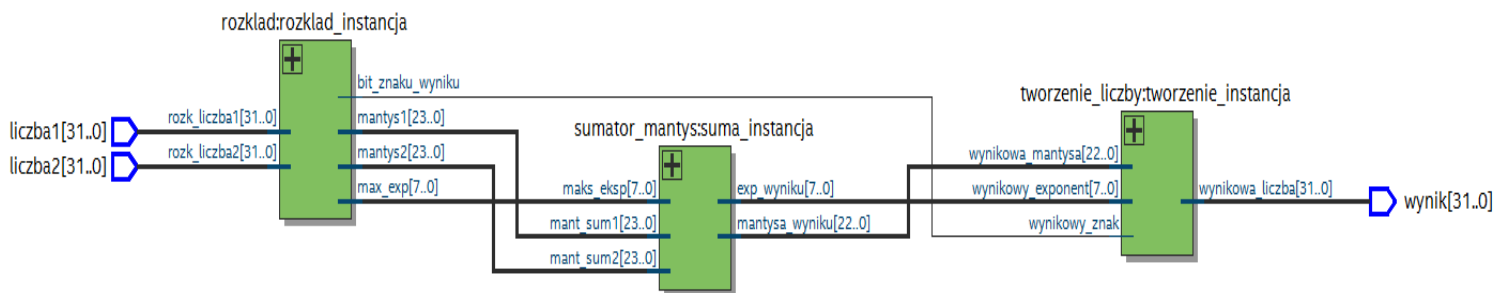
```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4
5  entity tworzenie_liczby is
6  port (
7      wynikowa_mantysa : in std_logic_vector(22 downto 0);
8      wynikowy_exponent : in std_logic_vector(7 downto 0);
9      wynikowy_znak : in std_logic;
10     wynikowa_liczba : out std_logic_vector(31 downto 0)
11 );
12
13 end entity;
14
15 architecture main of tworzenie_liczby is
16     begin
17
18         process(wynikowa_mantysa, wynikowy_exponent, wynikowy_znak)
19
20             variable liczba_koncowa : std_logic_vector(31 downto 0);
21
22             begin
23                 -- połączenie 3 wynikowych elementów: znaku, cechy, mantysy
24                 liczba_koncowa := wynikowy_znak & wynikowy_exponent & wynikowa_mantysa;
25
26                 wynikowa_liczba <= liczba_koncowa;      -- wynikowa liczba jest 32-bitowa
27
28             end process;
29
30 end main;
```

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

6. Schemat blokowy wygenerowany w

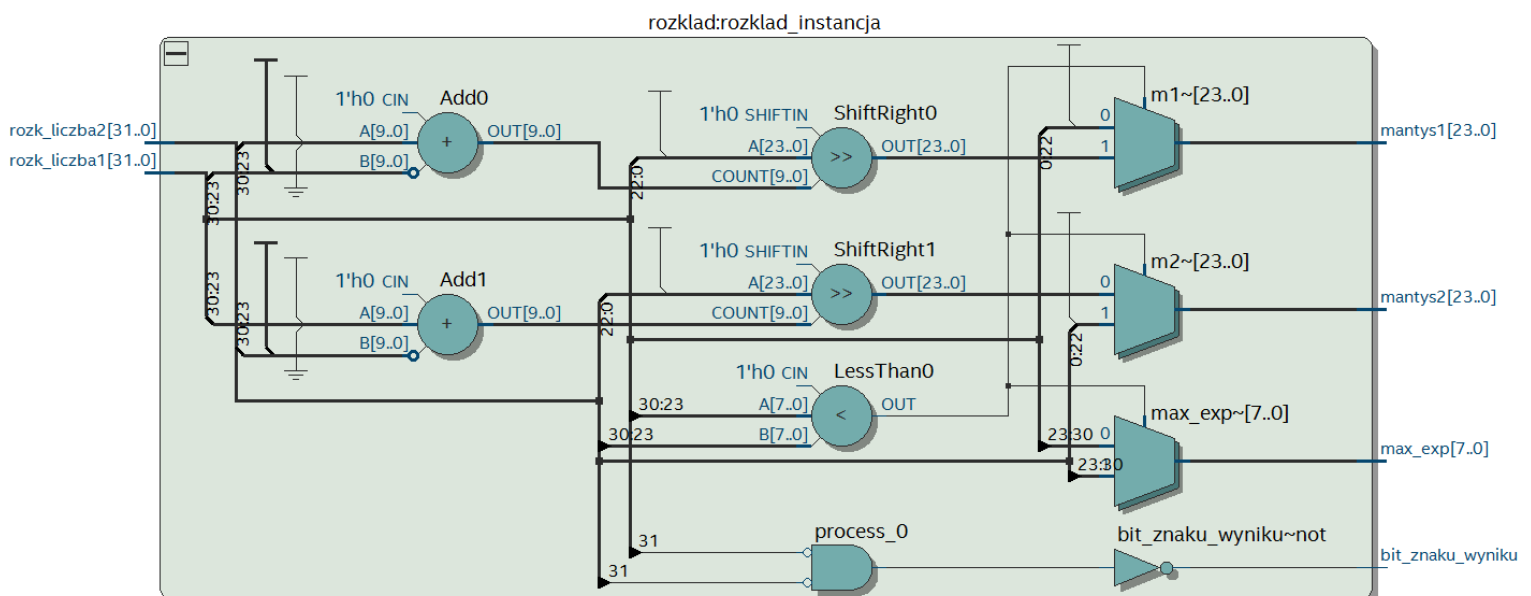
Quartusie:

- Schemat wygenerowany przez Quartusa pokrywa się z naszą wersją (stworzoną w formie rysunku) przed implementacją.(Punkt 4.)



- Przedstawimy wielkość pojedynczych komponentów pod względem wykorzystanych operatorów logicznych i innych operacji(pętli, ilości wykorzystanych zmiennych) w postaci użytych, prostych i złożonych, bramek logicznych.

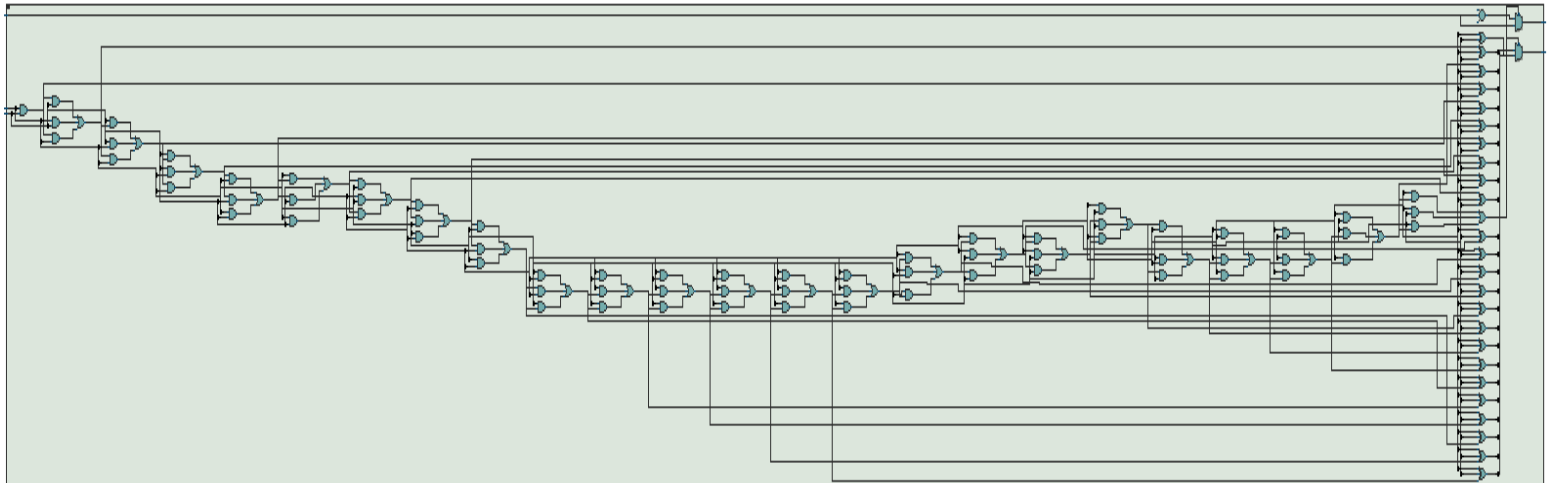
I. Rozkład liczb.



“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

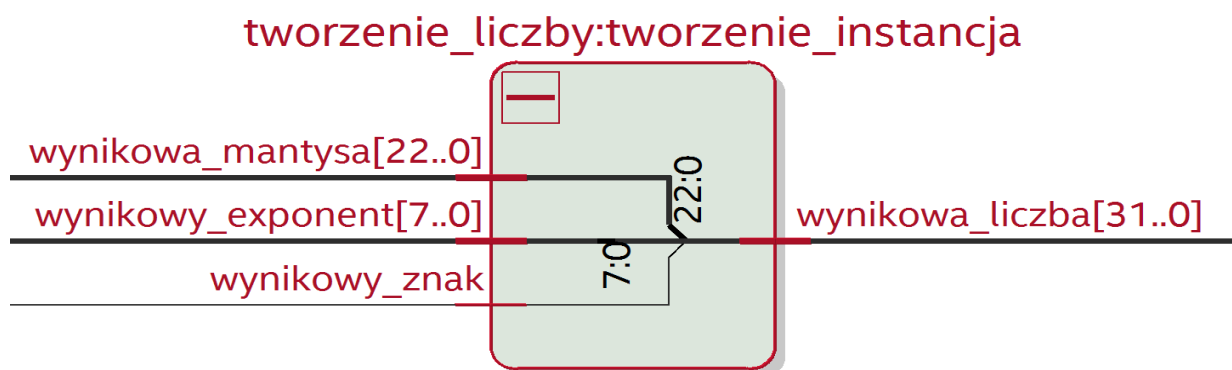
- Jak można zauważyć, ilość bramek logicznych nie jest ogromna, operacje wykonywane na potrzeby algorytmu w tym komponencie to: instrukcja warunkowa ‘if’, funkcja przesunięcia ‘shift’, operator ‘–’ dla zmiennych typu integer.

II. Sumowanie mantys



- Schemat na poziomie bramek logicznych tutaj już jest ogromny. Nie posiada on zbyt skomplikowanych operatorów logicznych jak poprzedni schemat, jednakże ich ilość jest duża. Wniosek z tego jest oczywisty, korzystanie z operacji typu ***if*** czy pętla ***for*** (zwłaszcza pętli!) powinno być ostatecznością, jeżeli zależy nam na szybkości działania programu.

III. Połączenie 3 elementów – wynik.



- W tym bloku operacją, było przypisanie wyjściowemu sygnałowi, odpowiedniego połączenia 3 elementów: **znaku** **cechy** **mantysy**. W związku z tym ten blok jest pusty.

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

7. Symulacje w Quartus Prime Lite.




- Symulacje wykonywane są przy pomocy pliku *.vwf – waveform’a. Generuje on za każdym razem testbench’a, dla naszego przykładu obliczeniowego.

1. Przykładowe liczby X,Y:

- $X = 0100\ 0010\ 0000\ 1111\ 0000\ 0000\ 0000\ 0000$
- $Y = 0100\ 0001\ 1010\ 0100\ 0000\ 0000\ 0000\ 0000$

Wynik Sumowania przeprowadzonego analitycznie:

$$\diamond W_1 = 0100\ 0010\ 0110\ 0001\ 0000\ 0000\ 0000\ 0000$$




		Value at 0 ps	480,0 ns	500,0 ns	520,0 ns	540,0 ns
	▷ liczba1	B 01000010000011110000000000000000	01000010000011110000000000000000			
	▷ liczba2	B 01000001101001000000000000000000	01000001101001000000000000000000			
	▷ wynik	B 01000010011000010000000000000000	01000010011000010000000000000000			

2. Przykładowe liczby A,B:

- $A = 0110\ 1011\ 1111\ 0011\ 1010\ 0000\ 1100\ 0011$
- $B = 0110\ 1011\ 1000\ 1110\ 0101\ 1111\ 0001\ 1100$

Wynik Sumowania przeprowadzonego analitycznie:

$$\diamond W_2 = 0110\ 1100\ 0100\ 0000\ 1111\ 1111\ 1110\ 1111$$

		Value at 0 ps	480,0 ns	500,0 ns	520,0 ns
	▷ liczba1	B 01101011111100111010000011000011	01101011111100111010000011000011		
	▷ liczba2	B 01101011100011100101111100011100	01101011100011100101111100011100		
	▷ wynik	B 01101100010000001111111111011111	01101100010000001111111111011111		




“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

3. Przykładowe liczby C,D:

- C = 0110 1011 1111 0011 1010 0000 1100 0011
- D = 0110 1000 1000 1110 0101 1111 0001 1100

Wynik Sumowania przeprowadzonego analitycznie:

❖ $W_3 = 0110\ 1011\ 1111\ 0101\ 1101\ 1010\ 0011\ 1111$

	Name	Value at 0 ps	480,0 ns	500,0 ns	520,0 ns
	▷ liczba1	B 011010111111100111010000011000011	011010111111100111010000011000011		
	▷ liczba2	B 01101000100011100101111100011100	01101000100011100101111100011100		
	▷ wynik	B 011010111111101011101101000111111	011010111111101011101101000111111		

“Sumator dwóch liczb zmiennoprzecinkowych o pojedynczej precyzji, w standardzie IEEE 754.”

Wnioski:

- Wyniki z 3 przypadków policzonych analitycznie w porównaniu do wyznaczonych z symulacji wyszły takie jak się spodziewaliśmy – identyczne.
- Najbardziej zależało nam na sprawdzeniu poprawności działania projektu oraz pokazaniu wygenerowanych struktur każdego z komponentów, na poziomie bramek logicznych i zaobserwowaniu podstawowych różnic, wynikających z ich implementacji - operacji które musiały wykonać.
- Zaprojektowany sumator działa prawidłowo
- Gdyby interesował nas czas kompilacji, można by zastanowić się nad zmniejszeniem ilości komponentów na schemacie blokowym i sprawdzić jakie wyniki czasowe otrzymalibyśmy.

Bibliografia:

- „Projektowanie układów cyfrowych z wykorzystaniem języka VHDL”
Mark Zwoliński
- <https://www.doulos.com/knowhow/>
- <http://rawski.zcb.tele.pw.edu.pl/category/ucyf-plansze/>
- https://eduinf.waw.pl/inf/alg/006_bin/0022.php