LISTA 2

ZAD. 1

1) $(x > 0) \,||\, (x - 1 < 0)$

Nie, ponieważ dla $x = -2$ prawa i lewa strona wyrażenia jest fałszywa.

$x = -2$ nie jest większy od 0, a $x - 1 = -2 - 1 = 2^{31} - 1$ zatem wynik nie jest mniejszy od 0.

$$\begin{array}{r} 100000\ldots000 \\ -\ 000000\ldots001 \\ \hline 011111\ldots111 \end{array}$$

3 max dodatnia

0 - 0 = 0
0 - 1 = 1 + 1
1 - 0 = 1
1 - 1 = 0

2) $(x \,\&\, 7) \,!= 7 \,||\, (x << 29 < 0)$ $\quad 7 = 111$

Tak, 7 w zapisie binarnym to 111, zatem jedyna sytuacja w której prawa strona wyrażenia jest fałszywa to taka gdy na ostatnich trzech pozycjach x będzie miał same jedynki. Podstawiając takiego x pod prawą stronę wyrażenia zawsze będzie ono prawdziwe, gdyż po przemnożeniu x będzie miał postać 110...0, a więc będzie liczbą ujemną.

3) $(x * x) \geq 0$

4) $(x < 0) || (-x \leq 0)$

Tak, rozważmy dwa przypadki:
- gdy $x<0$ to spełniona jest lewa część wyrażenia
- gdy $x \geq 0$ to spełniona jest prawa część wyrażenia gdyż |int32_t_MAX| < |int32_t_MIN| więc nie przekroczy zakresu.

5) $x > 0 || -x \geq 0$ Nie, dla x = INT32_MIN

Nie, rozważmy dwa przypadki:
- gdy $x>0$ to spełniona jest lewa część wyrażenia
- gdy $x \leq 0$ to zwróci fałsz, ponieważ |int32_t_MIN| > |int32_t_MAX|, czyli przekracza zakres.

6) $(x | -x) >> 31 == -1$

Nie, dla $x=0$, bo $0 >> 31 = 0$
(dla wszystkich innych jest spełniona)

7) ((uint32_t) x) >> 3 == ((uint32_t) x) / 8

$2^3 = 8$

Tak, ponieważ przesunięcie bitowe to to samo co dzielenie, dla liczb dodatnich (bez znaku)

8) $x >> 3 == x / 8$ $x = -1$ jak podzielić to powie że to 0, a przesunięcie nic nie zrobi

Nie, ponieważ dla liczb ujemnych (ze znakiem) przesunięcie w prawo spowoduje skopiowanie najbardziej znaczącego bitu na nowe pozycje. Wyrażenie będzie zwracało fałsz dla ujemnych liczb niepodzielnych (ze względu na zaokrąglenie przy dzieleniu)

8 nie wszystkie zapalone bity

9) x + y == (uint32_t) y + (uint32_t) x

Tak, podczas wykonywania operacji zmienne ze znakiem zostaną zmienione na zmienne bez znaku.

ZAD. 2
Prosty sposób na sprawdzenie czy $x<y$ to obliczenie $x-y$. Gdy ta wartość jest ujemna, to y jest większy, wpp. nie jest większy.
Niestety w ten sposób możemy wyjść od overflow np. INT_MIN-4 da nam liczbę dodatnią, a w tym przypadku $x<y$, więc mamy błąd. Rozwiązaniem jest podzielenie wartości x i y przez 2.
To tworzy kolejny problem, ponieważ tracimy bity jedynki w x i y.
PRZYPADKI:
A) x nie zawiera 1 i y nie zawiera 1: nic nie robimy, ponieważ nic nie tracimy

B) x nie zawiera 1 i y zawiera 1: odejmujemy od różnicy (~x & y & 1), ponieważ w tej sytuacji y mogło być większe od x, ale straciliśmy bit który o tym przesądził. Dokładniej (x >> 1) - (y >> 1) będzie równe 0, a my odejmiemy od tego 1 i dostaniemy -1

C) x zawiera 1 i y nie zawiera 1: nic nie robimy, ponieważ wtedy jeśli (x >> 1) - (y >> 1) będzie równe 0, to i tak x > y

D) x zawiera 1 i y zawiera 1: nic nie robimy, ponieważ wtedy jeśli (x >> 1) - (y >> 1) będzie równe 0, to x = y

Następnie przesuwamy bit znaku o 31 i jeśli mamy ujemną to w operacji & da nam 1

$(((x >> 1) - (y >> 1) - (\sim x \& y \& 1)) >> 31) \& 1$

ZAD. 3

b = x >> 31
return ~b & x + b & ~(x - 1)

~(x - 1) = x np.
x = 1001 (-7)
- 0001
~1000
0111 -> 7

dodat. b = 000...0	x		00000 0	= x
ujemne. b = 11...1	0000..0	+	-x	= -x
zero b = 00...0	x		00...0	= x

INNY
y = x >> 31
return (x ^ y) - y

ZAD. 4 (x >> 31) - ((-x) >> 31)

jeśli się pruje, że nie się nie zmieści to napisz uint32t y = -x i podstaw y pod -x

dla x < 0
1...1111 - 000....0 = 1111111111 = -1

dla x = 0
00000...0 - 000.... 0 = 000... 0 = 0

dla x > 0
000... 0 - 1..111111 = 000...01 = 1

111
0000
= 1111
0001

ZAD 5.

```
int32_t z5_inrange(int32_t x, int32_t y)
{
    int32_t z = x+y;

    int32_t a = x >>31;
    int32_t b = y >>31;
    int32_t c = z >>31;

    return ((a^c) & (b^c))+1;
}
```

3 przypadki:

- x,y i z mają taki sam znak, wtedy nie ma niedomiaru ani nadmiaru, więc zwraca 1
- x,y mają ten sam znak, a z ma inny wtedy mamy nadmiar lub niedomiar
- jeśli x i y mają różne znaki to nie ma nadmiaru ani niedomiaru

ZAD. 6

Na podstawie zad. 4 z listy 1

```
int32_t odd_ones(uint32_t x){

    x = (x & 0x55555555) + ((x>>1) & 0x55555555);
    x = (x & 0x33333333) + ((x>>2) & 0x33333333);
    x = (x & 0x0f0f0f0f) + ((x>>4) & 0x0f0f0f0f);
    x = (x & 0x00ff00ff) + ((x>>8) & 0x00ff00ff);
    x = (x & 0x0000ffff) + ((x>>16) & 0x0000ffff);
    return x & 0x00000001
}
```

zad. 7 zwraca liczbę, która mówi ile 1 występuje w x.
Ostatnie operacja sprawdza co znajduje się na ostatnim bicie tej liczby. Gdyż to mówi nam czy liczba jest parzysta czy nieparzysta.

ZAD. 8.

1) `x == (int32_t)(double)x`

TAK
1) double ma taką samą precyzję jeśli chodzi o liczby całkowite co int32_t

`x == (int32_t)(float)` → ma mniejszą precyzję

2) NIE, float ma 32 bity, ale mieści na nich liczby całkowite i ułamkowe

3) d == (double)(float) d

float ma mniejszą liczbę bitów, więc ma mniejszy zakres

4) f == (float)(double) f

Prawda

5) f == -(-f)

float w swojej reprezentacji trzyma minusa na najstarszym bicie. Znak "-" można zmienić znak na najstarszym bicie

Prawda

6) 1.0 / 2 == 1 / 2.0

Prawda, jeżeli chociaż jedna float to druga też zostanie przerobiona na float

7) d * d >= 0.0

Prawda

- jeżeli f jest bardzo duże, a d małe to f+d zaokrągli się do f i wtedy f-f=0 a d nie jest 0

8) (f + d) - f == d

Nie, ze względu na niedokładność obliczeń ~~liczb~~ na liczbach zmiennoprzecinkowych

ZAD. 7

Standard IEEE 754-2008 definiuje liczby zmiennoprzecinkowe o szerokości 16-bitów. Oblicz ręcznie $3.984375 \cdot 10^{-1} + 3.4375 \cdot 10^{-1} + 1.771 \cdot 10^{3}$ używając liczb w tym formacie. Zapisz wynik binarnie i dziesiętnie. Czy wynik się zmieni jeśli najpierw wykonamy drugie dodawanie?

ROZWIĄZANIE

Liczba zmiennoprzecinkowa 16-bitowa w standardzie IEEE 754-2008 składa się z 1 bitu znaku, 5 bitowego wykładnika oraz 10 bitowej mantysy. Bias wykładnika wynosi 15.

1. KONWERSJA

Zacznijmy od zamiany liczb do postaci wygodnej dla rozważanego sposobu zapisu liczb zmiennoprzecinkowych.

$3.984375_{10} \cdot 10^{-1} = 0,3984375_{10} = 0,0110011_2 = 1,10011_2 \cdot 2^{-2}$

$\frac{1}{2} = 0,5 \quad \frac{1}{4} = 0,25 \quad \frac{1}{8} = 0,125 \quad \frac{1}{16} = 0,0625$

$0,3984375 = 0 * (1/2) + 1 * (1/4) + 1 * (1/8) + \ldots$

Ta liczba w formacie 16-bitowym IEEE 754-2008 zostanie zapisana następująco:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+/-	WYKŁADNIK					MANTYSA									
0	0	1	1	0	1	1	0	0	1	1	0	0	0	0	0

Podobnie postępujemy z pozostałymi liczbami

$3.4375_{10} \cdot 10^{-1} = 0,34375_{10} = 0,0101112 = 1,0111_2 \cdot 2^{-2}$

$1.4771_{10} \cdot 10^{3} = 1101110101112 = 1.1011101011_2 \cdot 2^{10}$

NALEŻY SPRAWDZIĆ CZY:

- część ułamkowa mieści się w 10-bitowej mantysie (tak, w szczególności ostatnie liczba w całości wypełnia mantysę)
- wykładnik zawiera się w przedziale -14..15 (tak)

2. OBLICZENIA

Dodawanie liczb zmiennoprzecinkowych odbywa się przez zrównanie położeń separatora części całkowitej i ułamkowej, wykonanie działania bez straty (dodatnie) i ostatecznie konwersji do użytego formatu

$(1.1001_2 \cdot 2^{-2} + 1.011_2 \cdot 2^{-2}) + 1101110101112 \cdot 2^{10} =$

$= 1.011111_2 \cdot 2^{-1} + 1.1011010112 \cdot 2^{10} =$

$= 1101110101.0110011_2$

```
  0 . 0 1 1 0 0 1 1
+ 0 . 0 1 0 1 1 0 0
-------------------
= 0 . 1 0 1 1 1 1 1
```

```
  1 1 0 1 1 1 0 1 0 1 1 . 0 0 0 0 0 0 0
+ 0 0 0 0 0 0 0 0 0 0 0 . 1 0 1 1 1 1 1
---------------------------------------
= 1 1 0 1 1 1 0 1 0 1 1 . 1 0 1 1 1 1 1
                          G . R X X X X X
```

Mantysa mieści jedynie tylko 10 bitów - musimy dokonać zaokrąglenia. Bit G (ostatni w mantysie) wynosi 1, R (pierwszy poza) wynosi 1 zatem zaokrąglenie wykonamy w górę.

Gdyby zmienić kolejność dodawania $1.1001_2 \cdot 2^{-2} + (1.011_2 \cdot 2^{-2} + 1.1011010112 \cdot 2^{10})$ pierwsze dodawanie przebiegło by następująco

```
  1 1 0 1 1 1 0 1 0 1 1 . 0 0 0 0 0 0 0
+ 0 0 0 0 0 0 0 0 0 0 0 . 0 1 0 1 1 0 0
---------------------------------------
  1 1 0 1 1 1 0 1 0 1 1 . 0 1 0 1 1 0 0
                          G . R X X X X X
```

W tej sytuacji G=1, R=0, S=1 - wykonujemy zaokrąglenie w dół