From:	Lukasz Tomaszek
To:	psokolnicki@interia.pl;
CC:	
Subject:	iee 754
Date:	25 stycze• 2006 17:08:13
Attachments:	ATT00005.txt

Co student PWR o IEEE 754 wiedziec powinien

0. Wstep

To bedzie szybka manual na temat IEEE 754, dzialan i nawiazan do rzeczywistosci (systemu dziesietnego). Nikomu nie bede tlumaczyl zapisu liczby w U2, bez znaku czy innych, prozaicznych rzeczy. Rozsadne pytania i sugestie mile widziane. Jesli nie rozumiesz prostych zapisow, to dla Ciebie juz nie ma nadziei.

Tekst pisany jest 'bez polskich znaczkow'. Byc moze kiedys sie to zmieni.

ChangeLog:
18/19 marca 2003 - wersja 0.8 RC1
Pierwsza poprawna wersja publiczna. Tekst nie jest jeszcze skonczony. Wersja 'przymusowa' zawierala wiele bledow i sprzecznosci, ktore staralem sie poprawic.
14 marca 2003 - wersja 0.2ß
Pierwsza (przymusowa) wersja publiczna. Strescilem rozdzialy 2 i 3. Sypnal mi sie komp i rozdzialy 4 i kawalek 3 i 5
12 marca 2003 - wersja 0.1ß
Pierwsza wersja. Jedynie rozdzialy 0 i 1.

ToDo:
- Dodawanie i odejmowanie (w razie koniecznosci **czytajcie archiwum InfEki**)
- Zaokraglanie do nieparzystej
- Obliczanie bledu sredniego zaokraglenia
- Polskie znaki w tekscie i poprawa bledow ortograficznych/gramatycznych
- Uzupelnienia

1. Liczba

Liczba zmiennoprzecinkowa pojedynczej precyzji (**single** - nie mylic z liczba podwojonej precyzji - double, o tych samych, zreszta, czesciach) sklada sie ze **znaku**, **mantysy** i **wykladnika**. Znak przyjmuje wartosc **0** dla liczby dodatnej i **1** dla liczby ujemnej. Znak oznaczac bedziemy **Z**. Mantysa to cyfry po przecinku, przed ktorymi domyslnie stoi jedynka (nie zawsze - o tym dalej). Mantyse zapisujemy na **23** bitach. Przyklad mantysy i co ona oznacza:

```
          / ----- 23 bity ----- \
Mantysa   10101010101010101010101
Oznacza 1,10101010101010101010101

Mantysa   00000000000000000000000
Oznacza 1,00000000000000000000000
```

Warto zaznaczyc, ze jest to liczba **bez znaku** o ewentualnej ujemnosci informuje nas znak kodowany osobno. Dlatego tez same zera w mantysie to zera po przecinku i **1** przed przecinkiem, czyli w systemie dziesietnym jeden (ta jedynka na poczatku nie oznacza liczby ujemnej jak w U2; jesli to Cie konfuduje - dopisz 0 na poczatek). Mantyse oznaczali bedziemy **M**. Ostatnia 'czescia skladowa' liczby zmiennoprzecinkowej jest wykladnik. Oznaczac bedziemy go **W**. Wykladnik zapisywany jest na **8** bitach w systemie **+127** (z obciazeniem 127). Oznacza to, ze po odczytaniu wartosci liczby zapisanej na 8 bitach jako liczby bez znaku by uzyskac wartosc wykladnika trzeba odjac 127 (bo liczba jest obciazona o wartosc 127). Przyklady:

```
Wykladnik   00000010
Dziesietnie        2
Wartosc W   2 - 127 = -125

Wykladnik   01111111
```

```
Dzisietnie       127
Wartosc W   127 - 127 = 0

Wykladnik   10000111
Dziesietnie      135
Wartosc W   135 - 127 = 8
```

Czyli wykladnik moze osiagac wartosci od -127 do +128.

Aby poznac wartosc liczby w IEEE754 nalezy wszystko to zebrac do kupy. Oznacza to, ze trzeba uwzglednic znak a mantyse pomnozyc przez 2 do potegi wykladnik. Przyklad:

```
  zawartosc               oznacza
Z 0                       +1
M 10101010101010101010101  1,10101010101010101010101
W 01111111                 0
L = +1,10101010101010101010101 * 2^0 = +1,10101010101010101010101


  zawartosc               oznacza
Z 0                       +1
M 00000000000000000000001  1,00000000000000000000001
W 10000001                 2
L = +1,00000000000000000000001 * 2^2


  zawartosc               oznacza
Z 1                       -1
M 00000000000000000000000  1,00000000000000000000000
W 00001000                 -119
L = -1,00000000000000000000000 * 2^-119 = -1 * 2^-119
```

Oczywiscie w prawdziwych maszynach nie przechowujemy wszystkich tych wartosci osobno. Liczba bitow, na ktorych mozna przechowac liczbe zmiennoprzecinkowa wg IEEE754 to 32 (bo 1+8+23). Kolejnosc zapisu wartosci takrze jest ustalona i stanowi **ZWM** (ZWM jak Zwiazek Walki Mlodych - taki twor; latwo zapamietac). Przykladowe liczby:

```
      1   1
( 1 + - + - ) * 2
      2   8
|Z|        W        |                      M                      |
 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    /  |  \
                   /   |   \
                  /    |    \ te miejsca maja wage:
                 1/2  1/4  1/8 itd. Dlaczego? Architektura Komputerow sie klania

-256 (czyli -1*2^8)
|Z|        W        |                      M                      |
 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Wspomnialem wczesniej o liczbach, ktore nie maja domyslnej **1** przed mantysa. Sa to tak zwane liczby zdenormalizowane (w przeciwienstwie do tych, ktore jedynke maja i mienia sie znormalizowanymi). Oprocz podzialu na liczby z(de)normalizowane mamy takze kilka wartosci specjalnych. Wartosci te to **0** (a raczej **+0** i **-0**), + i - nieskonczonosc (bedziemy je oznaczac **+inf** i **-inf**) oraz nieliczba (wartosc, ktora nie podpada pod nic powyzszego, np. wynik dzielenia 0/0) oznaczana **NaN**. Wszystkie powyzsze wartosci poznajemy po wartosciach mantysy i wykladnika. Obrazuje to ponizsza tabela:

W	**M**	Typ liczby
rozny od **0** i rozny od **MAX**	dowolna	Liczba znormalizowana
0	rozna od **0**	Liczba zdenormalizowana
0	**0**	**+ / -0** (znak zale¿ny od bitu znaku)
MAX	rozna od **0**	**NaN** (rozroznianie znaku nie ma sensu)
MAX	**0**	**+ / -inf**

Pod pojeciem **MAX** rozumie same **1** (jedynki na wszystkich polach **M** lub **W** zaleznie od przypadku).

Podsumowujac. Jesli liczba zabiera w wykladniku same jedynki to jest nieskonczonoscia (jesli w mantysie sa same zera) lub nieliczba - wynikiem blednym (jesli w mantysa nie zawiera samych zer). Jesli liczba zawiera same zera zarowno w mantysie jak i w wykladniku (znak moze byc dowolny) to jest zerem. Jesli zawiera zera w wykladniku a niezera w mantysie to jest liczba zdenormalizowana postaci $(-1)^{z} * 2^{w-127} * 0,M$, a jesli ma wykladnik inny niz same zera lub same jedynki to jest liczba znormalizowana postaci $(-1)^{z} * 2^{w-127} * 1,M$.

2. Zaokraglenia

Radze przeczytac uwaznie, bo w poprzedniej wersji byly tu bzdury!!!

W celu 'ulatwienia' sprawy mamy do dyspozycji kilka rodzajow zaokraglen: z. przez uciecie (truncation), z. do **+inf**, z. do **-inf**, zaokraglenie symetryczne do najblizszej parzystej, zaokraglenie symetryczne do najblizszej nieparzystej, zaokraglenie do najblizszej i zaokraglenie do zera. W standardzie IEEE 754 znajduja sie 'tylko' z. do **+ / -inf**, z. do **0** i z. do najblizszej, choc mozna znalezc zrodla, w ktorych stoi napisane, iz konieczne jest rozniez zaokraglenie do najblizszej parzystej lub zaokraglenie przez obciecie. Prosty wniosek: 'przyzwoitosc' nakazuje jednak znac wszystkie mozliwe metody. Bedziemy kolejno rozpatrywac przypadki od najlatwiejszych.

Zaokraglenie przez obciecie
Choc nie jest to potrzebne przy obcinaniu wyjasnie czym sa bity GRS. Ich nazwy to odpowiednio: Guard bit, Round bit i Sticky bit. Sa to trzy dodatkowe bity dodawane za mantysa w celu zaokraglania wlasnie. Wyglada to tak:

```
              / ----- 23 bity ----- \
Mantysa       mmmmmmmmmmmmmmmmmmmmmmm
Z 'nadmiarem' mmmmmmmmmmmmmmmmmmmmmmmGRS
```

Mowiac prosto: jesli po wykonaniu jakiegos dzialania (mnozenie, dodawanie) troche nam sie 'rozrosnie' mantysa, to te nadmiarowe bity znajda sie wlasnie na pozycjach GRS. Przy czym nie bedzie to do konca ot takie proste przeniesienie. Zalozmy, ze dzielimy mantyse przez 4. Dzielenie przez cztery jest (w systemie dwojkowym) rownowazne przesunieciu bitow o dwa pola w prawo, tak wiec:

```
               / ----- 23 bity ----- \
Mantysa        00010010000101011001011
Po przesunieciu xx00010010000101011001011 0

Mantysa        01010101010101010101010
Po przesunieciu xx01010101010101010101010 0
```

Jak widac dwa bity 'wskoczyly' na pozycje GR. A co z S i iksami na poczatku? Najpierw S. S jest suma logiczna bitow na prawo od GR. Jasne? Tak myslalem. Mowiac prosciej - jesli przesuwajac w prawo jakies bity nie zmiescily nam sie na pozycjach mantysy i pol GR i wsrod tych pol byla **przynajmniej jedna jedynka**, to w miejsce S wpisujemy jedynke. W przeciwnym przypadku wpisujemy tam zero. Kilka przykladow:

```
               / ----- 23 bity ----- \
Mantysa        00000000000000000000001 przesuwamy o jedna pozycje,
Po przesunieciu x0000000000000000000000100

Mantysa        01010101010101010101010 przesuwamy o piec pozycji,
Po przesunieciu xxxxx01010101010101010101(010), czyli
Po przesunieciu xxxxx01010101010101010101011

Mantysa        11100001111000011110000 przesuwamy o cztery pozycje,
Po przesunieciu xxxx1110000111100001111100(00), czyli
Po przesunieciu xxxx1110000111100001111000
```

A co z iksami? Poniewasz wykonujemy dzialania na liczbach znormalizowanych przed mantysa stoi domyslna jedynka. Z kolei jest to liczba dodatnia, wiec przed ta jedynka stoi nieskonczenie wiele zer. I tak tez wypelniamy:

```
                 / ----- 23 bity ----- \
Mantysa        1,00000000000000000000001 przesuwamy o jedna pozycje,
Po przesunieciu 0,10000000000000000000000100

Mantysa        1,01010101010101010101010 przesuwamy o piec pozycji,
Po przesunieciu 0,00001010101010101010101011
```

```
Mantysa         1,11100001111000011110000 przesuwamy o cztery pozycje,
Po przesunieciu 0,00011110000111100001111000
```

Teraz wracamy do zaokraglenia. Zaokraglenie przez obciecie polega na obcieciu czesci, ktora nie miesci sie na pozycji mantysy. Nie warto dalej tlumaczyc, gdzy nie ma czego. Przyklad:

```
                          / ----- 23 bity ----- \
Wartosc do zaokraglenia 0,10000000000000000000000100
Wartosc po zaokragleniu 0,10000000000000000000000

Wartosc do zaokraglenia 0,00001010101010101010101011
Wartosc po zaokragleniu 0,00001010101010101010101
```

Krotko mowiac: nie wazne co bylo w polach GRS - odcinamy to. Taki sposob zaokraglania nie jest jednak stosowany kiedykolwiek (choc byloby prawdopodobniej nam latwiej). Odciecie nie jest w ogole czescia standardu, jednak stare uklady arytmetyczne, ktore proklamowaly swa kompatybilnosc ze standardem IEEE 754 zanim zostal on jeszcze zatwierdzony, implementowany ja jako glowna (jesli nie jedyna) metode zaokraglania. Takie 'cacka' mozna nazwac starymi procesorami arytmetyki zmiennoprzecinkowej.

Zaokraglenie do +inf

To zaokraglenie przyjmuje rozna postac zaleznie od znaku liczby. Latwo to wytlumaczyc tym, ze liczba ujemna zblizajac sie w strone nieskonczonosci maleje w module (maleje jej wartosc z pominieciem znaku), podczas gdy liczba dodatnia rosnie. Ilustruje to ponizsza zaleznosc. Przedstawilem ja wedlug zasady: liczbe z ulamkiem dziesietnym zaokraglamy (tak jak w szkole, w zerowce) w gore (czyli w strone **+inf**):

```
Przed zaokragleniem:
      -10,75          0        7,66
---------*------------|---------*----------->

Po zaokragleniu:
         -10          0          8
----------*-----------|----------*---------->
```

Jak widac sprawdzily sie przewidywania: liczba ujemna zmienila swoja wartosc modulo, liczba dodatnia zwiekszyla. Podobnie dzieje sie przy zaokraglaniu liczb zmiennoprzecinkowych zgodnych z IEEE 754.
Jesli liczba jest ujemna to ucinamy jej nadmiarowa czesc (podobnie jak odbywalo sie to w przypadku poprzednio omawianego zaokraglania). Kiedy liczba jest dodatnia - zwiekszamy mantyse o 1/2 **ulp** (czyli dodajemy jeden na pozycji mniejszej niz najmlodsza pozycja mantysy). Pozycja ta to **G**. Zaokraglenie takie mozna stosowac jedynie gdy **G**=1, **R**=0, **S**=0 (choc **nie** jest to zalecane) a nalezy wykorzystywac, gdy program wywoluje zaimplementowana funkcje **ceiling** (czyli zaokraglanie liczby do najmniejszej liczby calkowitej nie mniejszej niz dana liczba). Przykladowo w przypadku **GRS** = 100, jesli dodamy jeden na pozycji **G** (gdzie juz jest **1**), to nastapi przeniesienie na pozycje starsza, czyli na pozycje najmlodsza mantysy (ta o wadze **ulp**). Ilustruje to przyklad:

```
          / ----- 23 bity ----- \
       M 00000000000000000000001100
+1/2 ulp .......................1..
-----------------------------------
       = 00000000000000000000010000

          / ----- 23 bity ----- \
       M 11111001111001110011010100
+1/2 ulp .......................1..
-----------------------------------
       = 11111001111001110011011000
```

W powyzszych przykladach przebieglo to bezstresowo. Co jednak, gdy mantysa zawiera same jedynki (a moze sie tak zdazyc)? Liczbe (po wykonaniu dzialania) nalezy przeskalowac, to znaczy tak zmienic jej mantyse i wykladnik, by przed mantysa znajdowala sie tylko domyslna jedynka. Pamietamy, ze przesuwajac mantyse o jeden bit w prawo nie zmieniamy wartosci liczby jesli inkrementujemy wartosc wykladnika i na odwrot (oczywiscie: nie zmieniamy wartosci, jesli jakis wazny, czyli zawierajacy jedynke, bit nie przeniesie nam sie na pola GRS). Ilustruje to przyklad:

```
            / ----- 23 bity ----- \
       M 1,11111111111111111111111100
+1/2 ulp 0,00000000000000000000000100
-----------------------------------
      = 10,00000000000000000000000000 przesuwamy o jeden bit w prawo,
         1,00000000000000000000000000 nalezy jeszcze dodac jeden do wykladnika!
```

Tak oto nasza liczba ponownie stala sie liczba znormalizowana.

Pozwole sobie jeszcze wrocic do tematu przeskalowywania. Otoz korzystajac z kilku prostych operacji (mnozenia, dodawania i przesuwana bitow) mozna udowodnic, ze:

```
M * 2^W =
M / 2 * 2 * 2^W =
M / 2 * 2^(W + 1) =
M >> 1 * 2^(W + 1) =
```

Gdzie **>>** to przesuniecie bitow w prawo o pewna liczbe bitow/pozycji (tutaj: 1).
Oczywiscie w powyzszym przykladzie (i w kilku kolejnych) piszac **M** mam na mysli mantyse z dodana juz na poczatku jedynka (a nie gola sekwenjce zer i jedynek wpisanych w pamiec). Dodatkowo tutaj zrezygnowalem z -127 w wykladniku w celu lepszego unaocznienia samego skalowania.

Zaokraglenie do -inf

Tutaj sytuacja jest podobna do zaokraglania do **+inf**. Jedynie przypadki sa 'odwrotne'. Otoz, jesli zaokraglimy liczbe ujemna w dol, to musimy dodac na pozycji G 1/2 **ulp** a jesli zaokraglamy liczbe dodatnia to stosujemy uciecie. Powod jest podobny do poprzedniego: liczba ujemna zaokraglona w dol zwieksza swoja wartosc w module, a liczba dodatnia zmniejsza:

```
Przed zaokragleniem:
      -10,75          0        7,66
---------*------------|---------*----------->

Po zaokragleniu:
       -11            0        7
--------*-------------|--------*------------>
```

Samo zaokraglanie przebiega bardzo podobnie, wiec ograniczе sie do dwu przykladow:

```
Liczba dodatnia:
          / ----- 23 bity ----- \
      M 1,00000000000111000011111100
Uciecie 1,00000000000111000100000000

Liczba ujemna (Z=1):
           / ----- 23 bity ----- \
       M 1,11111111111111111111111100
+1/2 ulp 0,00000000000000000000000100
-----------------------------------
      = 10,00000000000000000000000000 przesuwamy o jeden bit w prawo,
        1,00000000000000000000000000 nalezy jeszcze dodac jeden do wykladnika!
```

Zaokraglenie do zera

Z teoretycznego punktu widzenia ta metoda niczym nie rozni sie od zaokraglania przez obciecie. Dlaczego wiec jest 'czyms innym'? Poniewaz generuje inne wyniki w systemach, w ktorych podstawa nie jest **2**, lub znak zapisywany jest w alternatyzny sposob (tzn. nie jako osobny bit). To jedyny powod rozroczniania tych dwoch nazw. W przypadku IEEE 754 z. do zera wyglada doklanie tak, jak obciecie. Kiedy stosujemy? Przy niektorych funkcjach konwersyjnych (np. przejscie z float do integer, choc mozna stosowac inne metody - wybor powinien nalezec do uzytkownika).

Zaokraglenie do najblizszej

Czym jest zaokraglenie do najblizszej? Posluze sie analogia z systemu dziesietnego. 10,2 jest blizej 10 niz 11, wiec zaokraglamy do 10. 3,89 jest blizej 4 i tak tez zaokraglamy. Jesli ktos jest bystry to zauwazy, ze wystarczy dodac polowe wartosci na najmlodszej pozycji (zwanej ulp) i obciac czesc ulamkowa. I tak tez dziala to zaokraglenie:

```
           / ----- 23 bity ----- \
       M 1,10001000100010001000100011
+1/2 ulp 0,00000000000000000000000100
-----------------------------------
         1,10001000100010001000100111 =
         1,10001000100010001000100
```

```
          / ----- 23 bity ----- \
      M 1,11111111111110000000000101
+1/2 ulp 0,00000000000000000000000100
---------------------------------
        1,11111111111110000000001001 =
        1,11111111111110000000001
```

Zaokraglenie to stosowane jest jako domyslne z. w ukladach kompatybilnych z IEEE 754.

Zaokraglenie symetryczne do parzystej

Zaokraglenie to nie znajduje sie w specyfikacji IEE 754, jednak implementowane jest czesto w ukladach arytmetyki zmiennoprzecinkowej. Jego druga zaleta jest uwielbienie, jakim palaja wykladowcy uniwersyteccy do tego rozwiazania. Do zalet z. symetrycznego do parzystej nalezy zerowy sredni blad zaokraglenia (mowiac prosciej: jesli rozpatrzymy wszystkie mozliwe zaokraglenia - czyli dla kazdej kombinacji GRS - i obliczymy blad, czyli roznice miedzy liczba zaokraglona a niezaokraglona, to suma wszystkich bledow wyniesie zero; wynika to z faktu, ze przy tym zaokragleniu rownie czesto wystepuje niedomiar, co i nadmiar, tzn. znak wyniku roznicy liczby przed i po zaokragleniu bedzie czasem + a czesem -).
Technika zaokraglania ta metoda nie nalezy do prostych. W przypadku, gdy **RS** = 00 z. to wyglada bardzo prosto: jesli G = 1 (a tylko wtedy mozna mowic o zaokragleniu) dodajemy lub odejmujemy **1/2 ulp** zaleznie od wartosci najmlodszej pozycji mantysy (to znaczy jesli mamy zero, a zero jest liczba parzysta - odejmujemy, a gdy jedynke to dodajemy). Przyklad wyglada tak:

```
                        / ----- 23 bity ----- \
Wartosc do zaokraglenia 0,10000000000000000000000100
Wartosc po zaokragleniu 0,10000000000000000000000

Wartosc do zaokraglenia 0,10000000000000000000001100
Wartosc po zaokragleniu 0,10000000000000000000010
```

Zakladam, ze kazdy juz wie jak przebiega dodawanie **1/2 ulp**.
Pozostale przypadki nie sa juz takie przyjemne. Mozna tutaj wykorzystac jeden z dwu schematow przedstawionych w Arytmetyce Komputerow znanego nam wszystkim autora. Przedstawie ten, ktory jest nieco latwiej zapamietac (bo oba sa niezwykle podobne).
Wybieramy liczbe niemniejsza niz zaokraglana, ktora na najmlodszym bicie mantysy ma **1**. Liczba ta ma byc mozliwe zblizona do liczby zaokraglanej, np. jesli zaokraglamy liczbe konczaca sie na ...0**110** (trzy kolorowe bity to **GRS**) to bliska jej liczba jest ...1xxx (**GRS** nas nie interesuja - nie ma ich). Teraz obliczamy roznice: liczba zaokraglana - liczba przez nas wybrana. Tutaj roznica bedzie +0**001** (gdzie pierwsze zero ma wage **ulp**). Skoro tak szukana wartoscia bedzie wynosic tyle, ile wybrana przez nas licza. Dlaczego? Funkcja wyboru wartosci na podstawie roznicy wyglada nastepujaco (**Y** - liczba zaokraglana, **M** - liczba, ktora wybralismy):

```
M-ulp      - ulp <= Y-M <= - 1/2 ulp
M      - 1/2 ulp  < Y-M  < + 1/2 ulp
M+ulp  + 1/2 ulp <= Y-M <= + 1 ulp
```

Na tej podstawie mozna stworzyc tabele (a raczej: odtworzyc korzystajac ze wspomnianej Arytmetyki...) dla przypadkow, gdy **S** = 0 (to i tak 8 przypadkow i duzo pisania, a obrazuja niemal calosc zagadnienia):

Przed zaokragleniem	...0**00**	...0**01**	...0**10**	...0**11**	...1**00**	...1**01**	...1**10**	...1**11**
Po zaokragleniu	...0	...0	...0	...1	...1	...1	...1+1	...1+1

Notacja ...1+1 oznacza koniecznosc dodania jeszcze jedynki (nie jest to przypadek ...0, gdyz konieczne jest jeszcze dodanie tej wlasnie wartosci - zwiekszenie mantysy przez inkrementacje). Przyklad:

```
          / ----- 23 bity ----- \
Liczba  1,10101010101010101010101110
'Wybor' 1,10101010101010101010101
Roznica: +011
Opcja: 'Wybor' + ulp
Wynik   1,10101010101010101010101 + 1, lub inaczej:
Wynik   1,10101010101010101010110
```

Nic dodac, nic ujac.

Zaokraglenie symetryczne do nieparzystej

Zaokraglenie do nieparzystej dziala podobnie do powyszszego. Poki co pomine je. Dociekliwi moga wyprowadzic a ja sie (moze) wyspie.

3. Mnozenie

Mnozenie opiera sie na prostych zalozeniach:

```
Skoro:
( (-1)^Z1 * M1 * 2^W1 - 127) * ( (-1)^Z2 * M2 * 2^W2 - 127) = (-1)^Z * M * 2^W - 127

Zatem:
Z = Z1 + Z2
M = M1 * M2
W = W1 + W2 - 127
```

Przy czym w wypadku znaku nie nalezy sie przejmowac brakiem miejsca na zapisanie wyniku przy 1 + 1, bo jak wiadomo 'minus razy minus daje plus'. Mantyse wyniku w przypadkach koniecznych nalezy przeskalowac (i odpowiednio zaokraglic). Jesli wykladnik nie miesci sie w zadanym przedziale od -127 do 128 (oczywiscie po uwzglednieniu ewentualnych inkrementacji wyniklych z przeskalowania mantysy) musimy 'wyrzucic sytuacje wyjatkowa', czyli zasygnalizowac blad. W przypadku, gdy liczba jest wieksza niz ta, jaka mozemy zmiescic zwracamy **+/-inf**, gdy jest zbyt mala **+/-0**, lub jesli jestesmy zbyt skonfudowani - **NaN**. Niestety nie mam dostepu do dziewiczej formy dokumentu IEEE 754 (paradoksalnie zeby ja sciagnac trzeba byc czlonkiem IEEE; taki 'zaszczyt' maja na przyklad prof. Biernat i prof. Piestrak) wiec nie wiem, ktora z powyzszych opcji jest ta zalecana przez standard. Na koniec przyklad mnozenia:

```
L1 = |0|10000000|10000000000000000000000|
L2 = |0|10000001|10000000000000000000000|
Obliczamy zatem: 3 * 6 = 18 (dziesietnie: [1,5 * 2^1] * [1,5 * 2^2])

Mantysa:
                        11000000000000000000000
                      * 11000000000000000000000
-----------------------------------------------
   11000000000000000000000.....................
+ 11000000000000000000000.....................
-----------------------------------------------
10010000000000000000000000000000000000000000000, przecinek na 2*23 = 46 pozycji
od prawej:
10,01000000000000000000000000000000000000000000

Skalujemy:
M = 1,00100000000000000000000, pominê zaokr¹glenie, bo GRS = 000
wykladnik zmienia sie o 1 (o 1 bajt przesunelismy w prawo)

Wykladnik:
W = 128 + 129 - 127 (+1)= 131
Liczba: L = |0|10000011|00100000000000000000000|
L = (dziesiêtnie) + 1,125 * 2^131-127 = + 1,125 * 2^4 = + 1,125 * 16 = 18
```

Jak widac wynik jest jak najbardziej poprawny.

4. Dodawanie

Jutro. Moze...

5. Odejmowanie

Jutro. Moze...

6. Dzielenie

Nie ma i raczej nie bêdzie.

A. Dodatek

FAQ

Q. Co to s¹ liczby pojedynczej/podwojnej precyzji?
Liczby te znajduja swoj odpowiednik w C jako zmienne typu single (pojedyncza precyzja) i double (podwojna precyzja). Jak nazwa sama wskazuje - te drugie sa bardziej precyzyjne, czyli w praktyce zapisywane na wiekszej ilosci bitow. Liczby pojedynczej precyzji zapisywane sa na 32 bitach i maja 23(a raczej 24, ale o tym dalej) bitowa mantyse, podczas gdy liczby podwojnej precyzji maja 52(53) bitowa mantyse. By w pelni odpowiedziec: sa jeszcze liczny rozszerzone (Extended), w ktorych mantysy maja odpowiednio 31(32) i 63(64) bity.

Q. Dlaczego 24 bity mantysy w single a nie 23?
Obie wartosci sa poprawne. W zapisie liczby mantysa zajmuje (przykladowo) 23 bity, jednak mozna (i raczej powinno sie) mowic, ze ma 24, gdyz tyle faktycznie bitow potrzeba do jej zapisania jesli nie pomijamy domyslnej jedynki przed mantysa. Generalnie przy operacjach musi sie tez ona znalezc w sumatorze/multiplekserze, wiec powinno sie mowic 24. Jednak, jak doswiadczenie pokazuje, co cialo pedagogiczne to zwyczaj, a te ciala wola 23.

Q. Po co liczby zdenormalizowane?
Trudno powiedzic. Byc moze, by uwzglednic szczegolne przypadki wymagajace wiekszej precyzji? A moze dlatego, ze zostaly wolne kombinacje Mantysa/Wykladnik? Jaki jest powod nie ma znaczenia. Liczby te sa w standardzie i staly sie przyczyna opoznienia w jego zatwierdzeniu. Do dzis wiele ukladow arytmtyki zmiennoprzecinkowej nie ma bezposredniej implementacji tych liczb.

Q. Czy sa inne standardy zmiennoprzecinkowe?
Tak i to niestety kilka. Jest pokrecony zapis logarytmiczny i oczywiscie nowsza wersja standardu IEEE - IEEE 854. Ten ostatni implikuje miedzy innymi koniecznosc implementacji liczb zmiennoprzecinkowych o bazie 10 (nie tylko 2).

Q. Jak, czy i kiedy dokonujemy wyboru metody zaokraglania?
Niestety wszystko zalezy od jednostki, ktora obliczenia wykonuje. Standard nie okresla gdzie maja znajdowac sie informacje o aktualnie wykorzystywanym typie z. Jedni producenci umieszczaja bity charakterystyczne w slowie statusowym mikrokontrolera, inni w specjalnych rejestrach, jeszcze inni w ogole nie daja mozliwosci wyboru.

Kontakt z autorami

email: IronHand
email: Ryan O'Connell