uCore goes floating point

Klaus.Schleisiek at spacetech-i.com

As every good Forth programmer I despised floating point. Real men use fixed point. Until I was supposed to compute the following expression in the Merlin project, which needs to stabilise laser frequencies to 20ppm precision in order to hit the methane absorption maximum. That implies that the laser temperature has to be stabilised as well, using a peltier element as the actor and an NTC resistor as the sensor.

$$T = \frac{B}{\ln\left(R/r_\infty\right)} \qquad R = r_\infty e^{B/T}$$

Therefore, I was confronted with these equations to compute the temperature from the resistance and to find the resistance set point for a certain temperature. (r_∞ is a pre-computable constant.)

At first, I used Matlab to compute a 3rd order polynomial function that fits "reasonably well" in the temperature range of interest. Nonetheless it was not easy to get the scaling right using */. The precision was disappointing and to make things worse, the error distribution of the two functions were inconsistent.

This was the starting point to rething my resistance to floating point. As a motivation, I will show you what I ended up with to solve the problem:

```
&3892  float Constant B-factor
-&298  float Constant -T0
&10000 float Constant R0
&27300       Constant 0_degC

B-factor -T0 f/ R0 fln f+ fexp Constant R_lim

: R>T   ( Ohm -- degC*100 )
   float R_lim f/   fln   B-factor swap f/   &100 float f* integer 0_degC -
;
: T>R   ( degC*100 -- Ohm )
   0_degC + float &100 float f/  B-factor swap f/   fexp   R_lim f*   integer
;
```

That's the code needed for the conversion functions that have fixed point numbers as inputs and outputs, scaled to Ohm and centidegC. To add even more to the motivation: All the floating point code needed cross-compiles into just 500 instructions (bytes).

Design principles

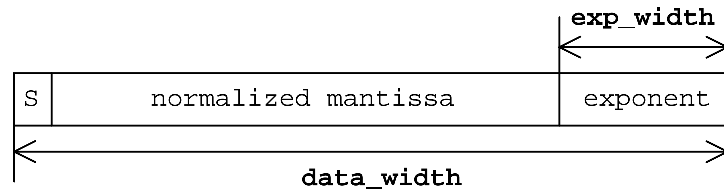
uCore has a configurable data word width and of course, the floating point representation must be able to cope with it. Therefore, IEEE-754 serves as a guideline and interchange format, but not as an implementation standard.

Nowadays, even on uCore, the **data_width** is at least 24 bits wide. Therefore, floating point numbers will fit on the stack. That already saves the code for a separate floating point stack.

Real number string input and output: Having written a floating point package for the RTX-2000 some 20 years ago, I remembered that about a third of the code dealt with proper number input and output. Not desirable. I have chosen a much simpler solution: integers can be converted to floating point using FLOAT, floating point numbers to integers using INTEGER. The floating point numbers can be properly scaled to engineering values using KILO, MEGA, MILLI, and MICRO in such a way that they properly scale using standard Forth number input and output.

Floating point representation

Floating point numbers are characterised by **exp_width**, the width of their exponent field. Whereas **data_width** is a more general parameter that specifies the native cell size of a uCore instantiation.



Exponent: I chose to put the exponent at the far right in order to cope with the variable data width easily. For an exp_width of 8 bits IEEE-854 adds a bias value of $7F to the exponent. That involves an add. uCore is just flipping the sign of the exponent using $80 xor, which consumes fewer logic resources.

Mantissa: IEEE-854 stores the mantissa as an absolute value preceeded by its sign. No rational is given but I suppose it has to do with the representation of + and - zero. It also avoids the singularity of the most negative number of 2s-complement representation. You don't know what I mean? - Just do "$80000000 abs u." in any 32-bit Forth. But computing the absolute value also involves an add and carry propagation and therefore, uCore stores the mantissa in 2s-complement representation.

I am still not sure whether there are more important reasons for IEEE's choices, but I have not seen any numerical misbehaviour due to uCore's non-standard representation.

Zero: uCore also has a positive and a negative zero. Given above choices, the positive floating point number zero is just - zero. Which is nice. When this zero has its sign set, it is a negative zero, making good use of the $80000000 sigularity. The check for floating zero is simple:

```
: f0= ( real -- flag )  2* 0= ;
```

There are only very few situations where the negative zero has to be explicitly handled and therefore, I believe it is a good choice.

Over/underflow: On overflow, the ovfl status bit will be set, and the largest positive or negative number will be returned depending on the expected sign of the result. On underflow, the unfl status bit will be set, and a positive or negative zero will be returned. For simplicity, there are no NaNs (Not-a-Number).

Hardware support

Four words have been implemented as uCore instructions for speed of execution:

`*. ( n u -- n' )`

It is used to compute mathematical functions based on polynomial expressions according to Horner's scheme: $(...((c_n * x + c_{n-1}) * x + c_{n-2}) * x + ... + c_0)$.

`normalize ( man exp -- man' exp' )`

The mantissa and the exponent are on the stack, both as 2s-complement numbers. Normalize shifts the mantissa to the left until only one single "leading" sign bit remains. The exponent is adjusted accordingly. This instruction can take several cycles depending on the magnitude of the mantissa. It gave rise to a uCore invention: Interruptible auto repeat instructions, which are simplifying uCore's instruction set considerably: um/mod, m/mod, um*, m*, sqrt, log2, shift, ashift can now be implemented as single instructions.

`>float ( man exp -- real )`

Amalgamates the mantissa and the exponent on the stack into a real taking care of the configurable floating point format. It also takes care of over/underflows, because the 2s-

complement exponent on the stack may not fit into the exponent field. The leading digit of a normalized mantissa will be dropped, because by definition its value is the inverted sign bit.

```
float> ( real -- man exp )
```

Converts the configurable real number into the mantissa and the exponent, both as full 2s-complement numbers.

Host support

I want to be able to compute floating point numbers "on the fly" during compilation. Therefore, a matching set of floating point words must be present on the host gforth system to support the uCore cross-compiler.

It is implemented in such a way that it takes care of the potential cell size difference of the host and the target. The host is characterized by constant **`cell_width`**, which can be determined automatically, and the target is characterized by constant **`data_width`**.

I have appended the code for gforth in the appendix and it can be downloaded at http://www.forth-ev.de/repos/microcore/trunk/Microcore/floating_point

Mathematical functions

Some functions can be computed using bit step algorithms: These are - besides multiply and divide - square root and logarithm (see **`log2`**). In principle, the **`exp2`** function could be computed bit wise as well, but it needs to take the square root in each step and therefore, a polynomial approximation is more efficient.

The other functions will have to be approximated to sufficient precision. In general, a real number will be split up into an integer (before the decimal point) and a fractional part (after the decimal point) after appropriate scaling. Then the fraction will feed the approximation function and the integer part will be handled in a function specific way. Most functions can be approximated by polynomials, which are evaluated using Horner's scheme and the **`*.`** operator.

My bible for function approximations is "Computer approximations" by John F. Hart, Wiley&Sons. It discusses the methods needed and presents coefficient sets for different precisions.

In the implementation for **`exp2`** and **`sin`** I have used the original coefficients from Hart and an on-the-fly scaling scheme to adapt to different data_widths. More functions will be added as the need arises.

Numerical precision

It is possible to compile the gforth code for different **data_width** and **exp_width** settings. Using

```
: ntc-test ( -- )
   &1000 &50000 bounds DO  cr I . I r>t dup . t>r . &1000 +LOOP ;
```

we get a good impression how the numerical precision of the **`fln`** and **`fexp`** code degrades when cutting down on the data_width. A **data_width** of 23 and an **exp_width** of 6 still produces results, which are far better than the initial integer based 3rd order polynomial approximation. Smaller data_widths damage the exp2 function, a smaller exp_width can not cope with the dynamics of the expressions any more.

This is a satisfying result, because in small systems I am usually using a 24 or 27 bit data_width.

Immenstaad, 1-Oct-2015

```
Only Forth also definitions

: shift  ( n1 quan -- n2 )   dup 0< IF  abs rshift  EXIT THEN  lshift ;
: ashift ( n1 n2 -- n3 )     dup 0< IF  negate 0 DO 2/ LOOP EXIT THEN  0 ?DO 2* LOOP ;
: u2/    ( u1 -- u2 )        1 rshift ;
: 2**    ( n -- 2**n )       1 swap lshift ;
: m/mod  ( d n -- rem quot ) fm/mod ;
: \\     ( -- )              source-id IF  BEGIN  refill 0= UNTIL  THEN  postpone \ ;

: cell_width ( -- u )  \ cell width of the host Forth system
   0 1 BEGIN  swap 1+ swap  2* ?dup 0= UNTIL
;
&32                         Constant data_width  \ cell width of target system
8                           Constant exp_width   \ width of exponent field
cell_width data_width -     Constant delta_width \ cell width >= data width !

data_width 1- 2**           Constant #signbit
exp_width 2** 1-            Constant #exp_mask
#exp_mask invert            Constant #man_mask
#exp_mask 2/ invert         Constant #exp_min
#exp_mask #exp_min and      Constant #exp_sign
#signbit                    Constant #fzero_neg
0                           Constant #fzero_pos
#signbit #exp_mask or       Constant #fmax_neg
#signbit invert             Constant #fmax_pos
-1 delta_width negate shift Constant #data_mask

Variable underflow  0 underflow !
Variable overflow   0 overflow !

Variable Scale  \ used for optimal scaling of a set of polynomial coefficients
: scaled     ( n -- n' )  s>d data_width 1 - 0 DO  d2*  LOOP Scale @ fm/mod nip ;
: scale_factor  ( n -- )  Scale ! ;

: round   ( dm -- m' )
   over 0< 0= IF  nip  EXIT THEN   \ < 0.5
   swap 2* IF  1+  EXIT THEN       \ > 0.5
   dup 1 and +                     \ = 0.5, round to even
;
: *.  ( n1 u -- n2 )  over 0< IF  swap negate um* round negate  EXIT THEN  um* round ;

: normalized?  ( m -- f )   dup #signbit and swap #signbit u2/ and 2* xor ;

: normalize    ( m e -- m' e' )
   over normalized? ?EXIT
   over 0= IF  drop   #exp_min  EXIT THEN
   BEGIN  dup #exp_min = ?EXIT
          1 -   swap 2* swap   over normalized?
   UNTIL
;
: >float  ( m e -- r )
   overflow off   underflow off
   normalize   swap #man_mask and swap
   over #fzero_neg =   over #exp_min =   and >r
   over #fzero_pos =   r> or
   IF  drop  #exp_mask invert and  EXIT THEN    \ leave floating +/-zero.
                                                \ For +zero irrespective of exponent
   dup #man_mask 2/ and
   dup 0< IF  #man_mask 2/ xor  THEN            \ exponent over/underflow?
   IF  0< IF  underflow on   0< IF  #fzero_neg  EXIT THEN  #fzero_pos  EXIT
        THEN   overflow on   0< IF  #fmax_neg   EXIT THEN  #fmax_pos   EXIT
   THEN
   dup #exp_min =
   IF  drop #man_mask and  EXIT THEN     \ smallest exponent => denormalized
   #exp_mask and   #exp_sign xor   swap  \ flip sign of exponent => bias = #exp_min
   dup 2* [ #signbit invert #exp_mask invert and ] Literal and
   swap 0< IF  #signbit or  THEN  or
;
```

```
: float>  ( r -- m e )
   dup #exp_mask and   ?dup 0= IF  #exp_min  EXIT THEN   \ de-normalized
   dup #exp_sign and IF  #exp_mask 2/ and
                     ELSE  #exp_mask 2/ invert or
                     THEN  swap                          \ flip sign and extend
   dup 0< IF  #exp_mask 2/ or  2/                        \ add 0.5 for rounding
          [ #signbit #exp_sign or u2/ invert ] Literal and
        ELSE  #man_mask   and u2/                        \ add 0.5 for rounding
          [ #signbit #exp_sign or u2/ ] Literal or
        THEN  swap
;
: int.frac  ( r -- frac int )  \ split float number into integer and fractional part
   float> [ data_width 2 - ] Literal +
   dup 0< IF  invert 0 ?DO  u2/  LOOP  2* 0  EXIT THEN
   0 swap [ delta_width 2 + ] Literal + 0 DO  d2*  LOOP
;
data_width &32 = [IF]

: >ieee ( r -- ieee )  \ only valid for 32-bit data_width
   float> $80 xor $7F + $FF and                   \ exponent
   over 0< IF  $100 or  THEN  &23 shift swap      \ sign
   abs -&7 shift $7FFFFF and or                   \ mantissa
;
: ieee> ( ieee -- r )  \ only valid for 32-bit data_width
   dup   dup 0< IF  negate $7FFFFF and $1000000 or
             ELSE  $7FFFFF and $800000 or
             THEN  7 shift
   swap -&23 shift $7F -  dup $80 and IF  $7F and  ELSE  $7F invert or  THEN  >float
;
[THEN]

: f+   ( r1 r2 -- r3 )
   float>   rot float>   rot 2dup -                              \ m2 m1 e1 e2 e1-e2
   dup 0< IF  swap >r nip  ELSE  rot >r nip >r swap r> negate  THEN \ m> m< diff_e1-e2
   1- dup [ data_width exp_width - negate ] Literal u< IF  drop 0 swap  THEN
   over IF  ashift  ELSE  drop  THEN  swap 2/ +   r> 1+ >float
;
: f*   ( r1 r2 -- r3 )
   float>   rot float>         \ m2 exp2 m1 exp1
   rot + data_width + -rot     \ exp3 m2 m1
   m* delta_width 0 ?DO  d2*  LOOP
   nip swap >float
;
: f/   ( r1 r2 -- r3 )   overflow off
   dup 2* 0= IF  invert xor #signbit and invert  overflow on  EXIT THEN
                                            \ leave +/- largest number on / by zero
   float>   rot float>
   data_width - rot - -rot
   0 swap delta_width 2 +  0 ?DO  d2/  LOOP
   rot m/mod nip   swap 2 + >float
;
: fnegate  ( r -- -r )
   dup 2* IF  float> 1+ swap 2/ invert #exp_sign 2/ + swap >float  EXIT THEN
   0< IF  0  EXIT THEN  #signbit               \ handle + and - zero
;
: fabs    ( r -- |r| )      dup 0< IF  fnegate  THEN ;
: f-      ( r1 r2 -- r3 )   fnegate f+ ;
: f<      ( r1 r2 -- f )    f- 0< ;
: f>      ( r1 r2 -- f )    swap f- 0< ;
: f<=     ( r1 r2 -- f )    f> 0= ;
: f>=     ( r1 r2 -- f )    f< 0= ;
: f0=     ( r -- f )        2* 0= ;
: f0<     ( r -- f )        0< ;
: f2*     ( r1 -- r2 )      float> 1+ >float ;
: f2/     ( r1 -- r2 )      float> swap 2/ swap >float ;

: float   ( n -- r )        0 >float ;
```

```
: integer ( r -- n )
   dup 2* #data_mask and 0= IF  2*  EXIT THEN  \ +/- zero
   1 float f2/ f+                               \ add 0.5 for rounding
   float> ashift
;
: 1/f     ( r1 -- r2 )      1 float swap f/ ;

: fscale  ( r1 n -- f2 )    dup 0< IF  abs float f/  EXIT THEN  float f* ;
: milli   ( r1 -- r2 )      -&1000 fscale ;
: micro   ( r1 -- r2 )      -&1000000 fscale ;
: kilo    ( r1 -- r2 )       &1000 fscale ;
: mega    ( r1 -- r2 )       &1000000 fscale ;

\ ***********************************************************************
\ logarithm, exponential
\ ***********************************************************************

: log2 ( frac -- log2 )   \ Bit-wise Logarithm (K.Schleisiek/U.Lange)
   delta_width 0 ?DO  2*  LOOP
   0   data_width 0
   DO  2* >r   dup um*
      dup 0< IF  r> 1+ >r  ELSE  d2*  THEN     \ correction of 'B(i)' and 'A(i)'
      round   r>                               \ A(i+1):=A(i)*2^(B(i)-1)
   LOOP  nip
;
: ?fzero ( r -- r / rdrop !! )  \ careful: manipulates rstack!
   dup 2* #data_mask and ?EXIT  drop #fmax_neg   overflow on   rdrop ;

: flog2  ( r1 -- r2 )  \ only defined for positive values
   ?fzero float> [ data_width 2 - ] Literal + 0 >float   swap
   abs 2* log2 u2/ [ data_width 1 - negate ] Literal >float f+
;
: exp2  ( ufrac -- uexp2 )
             \ Hart 1042, 23 bit precision, 1 > ufrac > 0, 1 = 2**(cell_width-1)
   [ &001877576 &008989340 + &055826318 +
     &240153617 + &693153073 + &999999925 + scale_factor ]
   >r [ &001877576 scaled ] Literal   r@ *.
      [ &008989340 scaled ] Literal + r@ *.
      [ &055826318 scaled ] Literal + r@ *.
      [ &240153617 scaled ] Literal + r@ *.
      [ &693153073 scaled ] Literal + r> *.
      [ &999999925 scaled ] Literal +
;
: +fexp2 ( r1 -- r2 )
   int.frac 2** float   swap exp2 [ data_width 2 - negate ] Literal >float f*
;
: fexp2  ( r1 -- r2 )   dup f0< IF  fnegate +fexp2 1/f  EXIT THEN  +fexp2 ;

&1442695 float micro Constant log2(e)

: fln  ( r1 -- r2 )  ?fzero flog2 log2(e) f/ ;

: fexp ( r1 -- r2 )  log2(e) f* fexp2 ;

\ ***********************************************************************
\ sine, cosine
\ ***********************************************************************

: sin ( ufrac --- usin )
           \ HART 3341  27 bit precision, pi/2 > frac >= 0, 1 = 2**(cell_width-2)
   [ &000151485 -&004673767 + &079689679 +
     -&645963711 + &1570796318 + 2* scale_factor ]
   dup >r  dup *. >r
   [  &000151485 scaled ] Literal   r@ *.
   [ -&004673767 scaled ] Literal + r@ *.
   [  &079689679 scaled ] Literal + r@ *.
   [ -&645963711 scaled ] Literal + r> *.
   [ &1570796318 scaled ] Literal + r> *.
;
&1570796327 float milli micro Constant fpi/2
```

```
: +fsin  ( r1 -- r2 )
   fpi/2 f/   int.frac >r
   r@ 1 and IF  invert  THEN  sin
   r> 2 and IF  negate  THEN
   [ data_width 2 - negate ] Literal >float
;
: fsin  ( r -- r' )  dup f0< IF  fnegate +fsin fnegate  EXIT THEN  +fsin ;

: fcos  ( r -- r' )  fpi/2 f+ fsin ;

: degree ( fdeg -- frad )   [ fpi/2 &90 float f/ ] Literal f* ;

\ ***************************************************************************
\ Converting NTC resistance to temperature and vice versa
\ ***************************************************************************

&3892  float Constant B-factor
-&298  float Constant -T0
&10000 float Constant R0
&27300       Constant 0_degC

B-factor -T0 f/ R0 fln f+ fexp Constant R_lim

: R>T   ( Ohm -- degC*100 )
   float R_lim f/   fln   B-factor swap f/   &100 float f* integer 0_degC -
;
: T>R   ( degC*100 -- Ohm )
   0_degC + float &100 float f/  B-factor swap f/   fexp   R_lim f*   integer
;
\ : ntc_test  ( -- )   &1000 &50000 bounds DO  cr I . I r>t dup . t>r . &1000 +LOOP ;
```