Forth Floating Point Word-Set
without Floating Point Stack

Willi Stricker

Springe, Germany,

September, 20, 2012

Preface

Originally computers work with integer numbers only. That is as well true for the Forth programming language with its basic words UM*, UM/MOD and further *, /, MOD, */ and */MOD.

For practical use this integer arithmetic is very restricted and there were lots of attempts to implement floating point arithmetic in computers as well. But in contrast to integer numbers there are lots of choices how to represent floating point numbers. Due to this problem the IEEE standards committee established the IEEE standard for floatng point numbers.

The main properties are:
The floating point number is separated in 3 parts: sign, significant and (transformed) exponent.
The significant with suppressed 1 (first bit) (normalized with a special denormalized treatment for very small numbers).
Especially they standardised two formats: single precision (32 bits wide) and double precision (64 bits wide) and additionally a special format for internal representation (80 bits wide).

Software versus hardware

At first the floating point arithmetic was programmed by software, using different formats. Later, hardware components were offered to avoid extensive programming and execution time, especially since the IEEE-format was established. The special floating point hardware is either an I/O-port or a so called „coprocessor“. Now the coprocessors are mostly incorporated in standard microprocessors.
The main features of floating point coprocessors are:

- Use of a special coprocessor interface with special instructions
- The internal representation of floating point numbers is 80 bits (IEEE special format)
- It has its own internal stack for floating point numbers
- The access of the operands needs special programming steps from „outside“ via the floating point stack.

Nowadays most „better“ microprocessors have an integratet floating point coprocessor. On the other hand especially for smaller systems without coprocessor it is sometimes necessary to use floating point numbers and arithmetic and that must be programmed by software.

Conseqence:
There are two choices:

- Using the floating point coprocessor directly, that means access of the operands by the floating point stack with special use of control instructions and control registers,
- Using floaing point arithmetic without floating point stack with either floating point created by software (no coprocessor) or using the coprocessor indirectly by hiding the hardware by software adaption.

In the Forth 2012 standard both choices are offered, implementation dependant.

Floating point arithmetic without floating point stack in Forth

That means the standard Forth parameter stack is used for all numbers in the same manner. For a 32 bit system: Integer and single precision floating point numbers have 32 bits, double integer and double precision floating point numbers have 64 bits

For example adding two numbers
with

```
VARIABLE INT1 VARIABLE INT2 VARIABLE INT3
VARIABLE FLO1 VARIABLE FLO2 VARIABLE FLO3
```
integer addition:
```
INT1 @ INT2 @ + INT3 !
```
floating:
```
FLO1 @ FLO2 @ F+ FLO3 !
```

Remark
There is no need for special treatment of fetching, storing or alignment of floating point numbers. Obviously the work with these numbers is much more easy to handle.

Problem: Double precision floating point numbers

The floating point numbers used in a floating point stack are always in 80 bit special format. Consequently there is no special difference necessary for single or double precision because all operations are done on the floating point stack. Only for input, output and storage it is necessary to convert the numbers into the used format.

In contrast the numbers on the parameter stack have to be in the working format. So all operations must be executed in that format.
Conclusion:
Like for integer numbers all operators must be present in single and double format (F+ and DF+ etc.).

For example adding two double precision numbers
with
```
2VARIABLE INT4 2VARIABLE INT5 2VARIABLE INT6
2VARIABLE FLO4 2VARIABLE FLO5 2VARIABLE FLO6
```
integer addition:
```
INT4 2@ INT5 2@ D+ INT6 2!
```
floating addition:
```
FLO4 2@ FLO5 2@ DF+ FLO6 2!
```

Double precision input

In Forth2012 floating point numbers have no distinguishing mark to indicate the input as single or double precision. For integers in Forth there is the mark dot „.“ to indicate the double precision input anywhere inside the digits.
Floating point numbers in Forth have a dot already as a position for the fractional (decimal) point. So other than Forth2012 standard **it is suggested to add another dot at the end of the input streem to indicate double precision floating point inputs.**
Examples:
single floating: 123.4E7 45E-6 -332E
double floating: 123.4E7. 45E-6. -332E.

Double precision output:

As stated above **outputs need a special word, suggested „D.“** The output is just printed with the maximum count of possible digits.

Examples:
input streem: 123.4E7
Result: FS. 1.234000E9
input streem: 123.4E7.
Result: DF. 1.234000000000000E9

Conclusion

Appendix A schows the whole Forth20012 floating-point word set in a table with remarks about the use with and without floating point stack and recommended words for double precision.

Appendix B shows the main Forth words for a floating point word set written in high level Forth to indicate that it is just a pretty simple and short program. It needs only about 2 kByte for a 32 bit system!

Appendix A

Comparison of floating point words with and without floating point stack with single and double precision:

Remarks:
*) Instructions necessary for double precision floating point arithmetic without floating point stack but not defined in Forth 2012
**) one of them maybe defined as standard, presumably FROUND

FORTH 2012 with Floating stack single and double precision	FORTH 2012 without floating stack single precision	FORTH 2012 without floating stack double precision

Format alignments

FALIGN	*Not used*	
FALIGNED	*Not used*	
DFALIGN		*Not used*
DFALIGNED		*Not used*
SFALIGN	*Not used*	
SFALIGNED	*Not used*	
FLOAT+	*Not used*	
FLOATS	*Not used*	
FFIELD	*Not used*	
SFLOAT+	*Not used*	
SFLOATS	*Not used*	
SFFIELD	*Not used*	
DFLOAT+		*Not used*
DFLOATS		*Not used*
DFFIELD		*Not used*

Declarations

FCONSTANT	Use CONSTANT instead	Use 2CONSTANT instead
FVARIABLE	Use VARIABLE instead	Use 2VARIABLE instead
FLITERAL	Use LITERAL instead	Use 2LITERAL instead
REPRESENT	*Not used*	
FVALUE	*Not used*	

Memory access

F@	*Use @ instead*	
F!	*Use ! instead*	
DF@		*Use 2@ instead*
DF!		*Use 2! instead*

Special control

FLOOR	*Not implemented*	
FROUND	*e.g. standard **)*	

FTRUNC	*e.g. standard **)*	
F~	*Not used*	
PRECISION	*Not used*	
SET-PRECISION	*Not used*	

Stack opertations

FDEPTH	*Use DEPTH instead*	
FDROP	*Use DROP instead*	*Use 2DROP instead*
FDUP	*Use DUP instead*	*Use 2DUP instead*
FOVER	*Use OVER instead*	*Use 2OVER instead*
FROT	*Use ROT instead*	*Use 2ROT instead*
FSWAP	*Use SWAP instead*	*Use 2SWAP instead*

Numerical output

F.	**F.**	
FE.	**FE.**	
FS.	**FS.**	**DF. *)**

Format conversion

>FLOAT	**>FLOAT**	
F>D	**F>D**	**DF>D*)**
F>S	**F>S**	
D>F	**D>F**	**D>DF *)**
S>F	**S>F**	
	F>DF *)	**DF>F *)**

Standard arithmetic functions

FNEGATE	**FNEGATE**	**DFNEGATE *)**
F+	**F+**	**DF+ *)**
F-	**F-**	**DF- *)**
F*	**F***	**DF* *)**
F/	**F/**	**DF/ *)**
FABS	**FABS**	**DFABS *)**

Comparison

FMIN	**FMIN**	**DFMIN *)**
FMAX	**FMAX**	**DFMAX *)**
F0<	**F0<**	**DF0< *)**
F0=	**F0=**	**DF0= *)**
F<	**F<**	**DF< *)**

Extended functions

FSIN	**FSIN**	**DFSIN *)**

FEXP	FEXP	DFEXP *)
etc	etc	etc ..... *)

Appendix B

Forth programs for a short floating point word set without floating point stack with single precision

Forth programs for the words FNEGATE, F+, F-, F*, F/, S>F, F>S written in high level forth

Explanations:
An intermediate format is defined for the three components, called „FX“ (s e m), that separates the components:
s = sign: msb. 0 = positive, 1 = negative,
e = exponent: signed exponent that is a retransformed transformed exponent (e = et – 7F),
m = mantissa: significant with open bit and normalized,
f = floating point number according to IEEE (single precision, 32 bits wide),
d = single integer number (32 bits wide)

```
********** floating kernel for 32 bit processor ****************
************************************************************
\ ---- conversion auxiliaries ---------------------------------
:    FX>F      ( m e s -- f )
[ HEX ] \ ----------------------------------------------------
 80000000 AND >R            \ sign
 7F + >R                    \ exponent
 DUP 0= IF R> DROP 0 >R THEN \ test 0 mantissa
 R@ 1 <
 IF                         \ denormalized ?
   R> 1- >R BEGIN U2/ R> 1+ DUP >R 0= UNTIL \ adjust exponent
 THEN                       \
 R@ FE > IF R> DROP FF >R DROP 0 THEN \ adjust infinity
 007FFFFF AND               \ clear hidden bit
 R> 17 SHIFTL OR            \ add exponent
 R> OR                      \ add sign
; DECIMAL

:    F>FX      ( f -> m e s )
[ HEX ] \ ----------------------------------------------------
 DUP 80000000 AND >R        \ sign
 DUP 7F800000 AND 17 SHIFTR 7F - >R \ exponent
 7FFFFF AND                 \ mantissa
 R@ -7F >                   \ not denormalized ?
 IF   800000 OR             \ add hidden bit
 ELSE DUP 0= NOT            \ else check for zero and normal.
   IF 2* BEGIN DUP 800000 AND 0= WHILE 2* R> 1- >R REPEAT THEN
 THEN                       \
 R> R>                      \ add exponent and sign
; DECIMAL

\ ---- fixpoint <-> floating point conversion -------------------

:    S>F       ( d -- f )
\ ------------------------------------------------------------
 DUP 0=                      \ zero ?
 IF
   -127 0                    \ e = -127, s = 0
 ELSE
   DUP [ HEX ] 80000000 [ DECIMAL ] AND >R \ sign
   DUP 0< IF NEGATE THEN
   31 >R BEGIN DUP 0< NOT WHILE 2* R> 1- >R REPEAT \ normleft
```

```
   8 SHIFTR R>               \ mantissa, exponent
   R>                        \ sign
 THEN                        \ m e s = fx
 FX>F                        \ convert to f
;

:    F>S       ( f -- d )
\ ----------------------------------------------------------
 F>FX                        \ m e s
 >R                          \ sign
 23 - DUP 7 >                \ exponent, 8-1 because of sign bit
 IF                          \ infinit )
   DROP DROP R> [ HEX ] IF 80000000 ELSE 7FFFFFFF THEN
   [ DECIMAL ]
 ELSE
   DUP 0>
   IF SHIFTL ELSE NEGATE SHIFTR THEN
   R> IF NEGATE THEN
THEN ;

\ ---- floating point arithmetic -----------------------------

:    FNEGATE    ( f -- fneg )
[ HEX ] \ ---------------------------------------------------
 80000000 XOR                \ toggle sign bit
; DECIMAL

:    F+         ( f1 f2 -- fsum )
[ HEX ] \ ---------------------------------------------------------
 >R F>FX R> F>FX             \ convert operands to fx
 >R 2 ROLL >R                \ save signs,
 2 ROLL OVER OVER MAX >R SWAP - DUP >R 0< \ exponent, exp-diff
 IF    SWAP R> NEGATE SHIFTR SWAP \ normalize m1 if necess
 ELSE  R> SHIFTR \ normalize m2 if necess
 THEN
 R> R> SWAP >R IF SWAP NEGATE SWAP THEN \ negate m1 if necess
 R> R> SWAP >R IF NEGATE THEN \ negate m2 if necess
 DUP 80000000 AND DUP R> SWAP >R >R \ add m1 and m2, sign
 IF NEGATE THEN               \ negate mantissa if necess
 DUP 0= NOT
 IF                           \ mantissa not zero ?
   DUP 00FFFFFF. U>
   IF
     U2/ R> 1+ >R
   ELSE                       \ normalize
     BEGIN DUP 800000 AND 0= WHILE 2* R> 1- >R REPEAT
   THEN
 THEN
 R> R>                       \ add exponent and sign
 FX>F                        \ convert to f
; DECIMAL

:    F-         ( f1 f2 -- fdiff )
\ ----------------------------------------------------------
 FNEGATE F+
;

:    F*         ( f1 f2 -- fprod )
[ HEX ] \ ---------------------------------------------------
 >R F>FX R> F>FX           \ convert operands to fx
 3 ROLL XOR >R             \ sign
 2 ROLL + >R               \ exponent
 UM*                       \ mantissa 64 bits
 FFFF AND WSWAP SWAP FFFF0000 AND WSWAP OR \ reduce to 32 bits
```

```
 DUP DUP 80 AND IF 100 + THEN \ round
 DUP 0<
 IF    SWAP DROP R> 1+ >R  \ normalize
 ELSE  DROP 2* DUP 80 AND IF 100 + THEN
 THEN                      \ normalize and round
 8 SHIFTR                  \ normalize to 24 bits
 R> R>                     \ add exponent and sign
 FX>F                      \ convert to f
; DECIMAL

:    F/         ( f1 f2 -- fquot )
[ HEX ] \ ------------------------------------------------------
 >R F>FX R> F>FX           \ convert operands to fx
 3 ROLL XOR >R             \ sign
 OVER 0= IF DROP DROP 800000 -96 THEN \ check for zero
 ROT SWAP - >R             \ exponent
 >R 0 SWAP R> UM/MOD2      \ mantissa division
 0> IF U2/ 80000000 OR ELSE R> 1- >R THEN \ normalize
 8 SHIFTR                  \ normalize to 24 bits
 SWAP DROP                 \ drop remainder
 R> R>                     \ add exponent and sign
 FX->F                     \ convert to f
; DECIMAL
```

Auxiliary instructions used in the above programs

```
: UM/MOD2 ( dl dh d – mod ql qh ) \ unsigned devide with double quotient result
 >R 0 R@ UM/MOD R> SWAP >R UM/MOD R> ;

: WSWAP ( w0:w1 – w1:w0 ) \ swap operand halphs
 DUP 16 LSHIFT SWAP 16 RSHIFT OR ;
```