Floating Point Math

Floating point numbers are handled by the C64's BASIC and Kernal ROMs. Called directly from machine language programs, they execute faster than when burdened by the BASIC interpreter. The difficulty lies in the variety of ways you must prepare the registers and zero-page before calling certain routines. Macros are recommended when dealing with FP in assembly.

Floating point numbers are stored with 5 bytes. The first byte is the exponent, stored in excess $80 format (E = E - $80). Second byte holds the sign of the mantissa in its uppermost bit. The normalized mantissa (0 =< M < 1) is assumed to have a leading 1, and the following bits hold the fraction. If the exponent is 0, the number is interpreted as 0 regardless of the contents of the mantissa.

81 00 00 00 00 = 1 = 2^1 x %1.000...
82 00 00 00 00 = 2 = 2^2 x %1.000...
82 80 00 00 00 = -2 =-(2^2 x %1.000...)
81 40 00 00 00 = 1.5 = 2^1 x %1.100...
82 40 00 00 00 = 3 = 2^2 x %1.100...

As you can see, multiplying or dividing a FP number by 2 is as easy as increasing or decreasing the exponent. When these numbers are brought to the ZP for use in the FP routines, the sign is extracted and replaced with the previously assumed 1, and the sign is stored in a separate byte.

Zero Page Locations

The FP routines make extensive use of the zero-page, so it is important to avoid using these areas to prevent errors. The first of the two most used locations is called the Floating Point Accumulator. The second is called the Floating Point Argument. Depending on the source, they are called either “FAC1” and “FAC2”, or “FAC” and “ARG.” After an operation, FAC1 will hold the result.

FAC1:

$61 holds the exponent
$62-$65 holds the mantissa
$66 holds the sign in bit 7

FAC2:

$69 holds the exponent
$6a-$6d holds the mantissa
$63 holds the sign in bit 7

The locations around the accumulators are used extensively too.

$56 is used in addition and subtraction, and also the EXP function.

$67 is used during series evaluations. BASIC uses series evaluations to calculate all the trigonometric

functions, the exponent functions, and the log functions. **$67** is also used during PETSCII to FP conversions.

$68 is used during normalisation, and also when converting a FP number to an integer (two similar operations).

$6f holds the sign result of arithmetic operations. This byte is set when FAC2 is loaded, not when FAC1 is loaded. FAC2 should be loaded after FAC1 before addition, or multiplication. Division does its own compare, so copying FAC1 into FAC2 and then loading FAC1 does not produce erroneous signs. Subtraction negates FAC1 and then compares, setting this byte.

$70 is the low order rounding byte for FAC1, essentially an extension of the mantissa. It is cleared whenever a variable is loaded into FAC1. Storing 0 here after obtaining a result can speed execution (at the cost of accuracy) by avoiding the rounding routines.

Four temporary accumulators exist as well:

$26-$2a is used to hold the result of multiplication or division. After the operation is concluded, this number is moved into FAC1. Safe to use if only using addition and subtraction.

$4e-$52 is used for BASIC's FN and $STR. Use freely.

$57-$5b is used when BASIC creates FP variables and arrays. Because you will be manipulating variables yourself, this area is free to use.

$5c-$60 is used by the PETSCII conversion routines, and like $57-$5b, for variables and arrays.

Several other locations are important depending on which routines you need:

$12 is used by the tangent function, but is also set (in preparation for TAN) by the sine and cosine routines.

$22/23 is a temporary vector often used by BASIC. For the FP routines, it is used to move variables into the accumulators, and back into memory. While you can use your own movement routines and free up this area a little, many of the more advanced routines make use of the standard Kernal moves. This vector is also used when evaluating RND.

$24/$25 is a temporary vector used during FP comparisons that points to the variable to compare to the accumulator. It is also used when converting PETSCII into FP.

$71/$72 is a temporary vector used during series evaluations, and when converting PETSCII into FP.

$73-$8a is the CHRGET subroutine. Copied to the ZP from RAM on startup, it contains a self-modifying vector that points to the next byte in a text line. It is used when converting a PETSCII string into FP. It is also used if you hook into BASIC's INPUT instruction to obtain an PETSCII string.

The Floating Point Routines

The routines that make up Commodore's FP package are as follows:

- Movement (Memory management)
- Basic arithmetic (Add, Subtract, Multiply, Divide)
- Trigonometry (SIN, COS, TAN, ATAN)
- Functions (Comparisons, SGN, ABS)
- Logarithms/Exponentiation
- Series Evaluations/Polynomials
- Type Conversion

The Trig, Log, and Exp routines, all use series evaluation to obtain their result, and all use multiply and divide. Their ZP locations become free to use if these routines are not needed. Likewise, the conversion routines consume another chunk of the ZP.

---

1. Movement

1.1 Load FP from memory

$bba2 = Load FAC1 from the 5 bytes pointed to by A/Y (low)/(high). Returns Y = 0 and A loaded with the exponent, affecting the processor status flags.
$ba8c = Load FAC2 from the 5 bytes pointed to by A/Y (low)/(high). Returns with FAC1's exponent in A.

It is possible to write your own version of these routines to use ZP direct addressing instead of indirect indexing.

1.2 Store FP in memory

$bbd4 = Stores FAC1 as 5 bytes at the address pointed to by X/Y (low)/(high). Checks the rounding byte ($70) beforehand, and rounds if the MSB is set. As with loading, it returns with the exponent in A.
$bcoc = Stores FAC1 in FAC2, rounding if necessary.
$bcof = Stores FAC1 in FAC2, skipping the rounding.
$bbfc = Stores FAC2 in FAC1.

Anything moving out of FAC1 is subject to rounding. When FAC1 is stored in FAC2, the rounded result is stored in both FACs. If you skip rounding, the rounding byte is still set to 0.

---

2. Basic Arithmetic

2.1 Addition

$b867 = Addition with memory contents pointed to by A/Y (low/high).
$b86a = Entry if FAC2 already loaded. The accumulator must load FAC1 exponent ($61) immediately before calling to properly set the zero flag.

Add FAC1 with FAC2, leaving the result in FAC1. Load FAC2 after FAC1, or else mimic the Kernal's sign comparison beforehand.

2.2 Subtraction

$b850 = Subtraction with memory contents pointed to by A/Y (low/high).
$b853 = Entry if FAC2 already loaded.

Subtract FAC1 from FAC2 leaving result in FAC1. Negates FAC1 and falls through to the addition routine.

2.3 Multiplication

$ba28 = Multiplication with memory contents pointed to by A/Y (low/high).
$ba2b = Entry if FAC2 already loaded. Accumulator must load FAC1 exponent ($61) beforehand to set the zero flag.

Multiply FAC1 with FAC2 leaving result in FAC1. Load FAC2 after FAC1, or else mimic the Kernal's sign comparison, then enter with the FAC1 exponent in A.

2.4 Division

$bb0f = Divide the memory contents pointed to by A/Y (low/high) by FAC1.
$bb12 = Entry if FAC2 already loaded. Accumulator must load FAC1 exponent ($61) beforehand to set the zero flag.

Divides FAC2 by FAC1, leaving the quotient in FAC1, and the remainder in FAC2.

2.5 Multiply FAC1 by 10

$bae2 = Multiply FAC1 by 10

Quickly multiplies FAC1 by 10. 2 is added to the exponent (x4). Then, the original value is added to the result (x5). Finally, the exponent is incremented again (x10).

2.6 Divide FAC1 by 10

$bafe = Divide FAC1 by 10

Not as quick as Multiply by 10. Shifts FAC1 to FAC2, loads FAC1 with the constant 10 from ROM, and falls through to the division routine. Note: This routine treats FAC1 as positive even if it is not.

---

3. Trigonometry

$e26b = Sine(FAC1)
$e264 = Cosine(FAC1)
$e2b4 = Tangent(FAC1)
$e30e = Arc-Tangent(FAC1)

All the trig routines, even though they only require FAC1 for input, trash FAC2. Enter all but ATAN with FAC1 expressed in radians. The sine routine uses the series evaluation routine to converge on a value. For cosine, the routine adds pi/2 (90 degrees) to FAC1, then falls through to the sine routine, because cos(x) = sin(x+(pi/2)). The tangent routine uses the fact that tan(x) = sin(x)/cos(x). ATAN uses its own series evaluation.

---

4. Functions

4.1 SGN

$bc2b = Evaulate sign of FAC1 (returned in A) **$bc39** = Evaulate sign of FAC1 (returned in FAC1)

These routines return $01 for positive numbers, $ff for negative numbers, and 0 for zeros. If you know a number is non-zero, it is quicker just to use BIT $66, or LDA $66, etc.

4.2 ABS

$bc58 = Returns absolute value of FAC1

Simply LSR's the sign byte of FAC1 ($66) to put a zero in the MSB. This is easily done without the need for a JSR.

4.3 INT

$bccc = Round FAC1 to Integer

Takes FAC1 and rounds away the decimal to make it an integer. It is still, however, expressed as a FP number.

4.4 Comparison

$bc5b = Compare FAC1 with memory contents at A/Y (lo/high)

Returns with A = 0 if the values are equal, 1 if FAC1 > MEM, or $ff if FAC1 < MEM. Uses the vector at $24 to address the variable, leaving FAC1 and FAC2 unchanged. If used in a loop, it's a good candidate for a rewrite, especially if you can compare the FAC to a location in the ZP.

---

5. Logarithms

5.1 Natural Log

$b9ea = Returns the natural log of FAC1

This returns the natural log of FAC1 in FAC1. The FAC1 can not be negative or zero, or else BASIC will respond with an ?ILLEGAL QUANTITY error. Calculated using a series.

5.2 EXP / e^x

$bfed = Returns e raised to the power of FAC1. (e^x : e = 2.718281828)

The opposite of log, i.e. log(exp(5)) = exp(log(5)) = 5, though the routine lacks the accuracy for this to always be true. Calculated using a series.

5.3 Exponentiation / y^x

$bf7b = FAC2 raised to the power of FAC1 (FAC2^FAC1)

This routine uses the formula exp(x*log(y)) to calculate y^x, so it calculates two series (log and exp). It is slow and not entirely accurate. For whole number powers, it is often quicker and more accurate to use a series of multiplies.

5.4 Square Root

$bf71 = Square root of FAC1
$bf74 = Square root of FAC2

The first address copies FAC1 into FAC2, before loading .5 into FAC1 and falling through to the exponentiation routine. The second address skips the move and uses the value in FAC2. A quicker square root routine is described below.

---

6. Series Evaluations

7. Type Conversion

7.1 FP to integer

7.2 FP to string

$bddd = Convert FAC1 to string at $100

The string returned is in PETSCII format, and is terminated with a zero byte. The PETSCII definition of a number does not differ from the screen code definition with the exception of the “E” when a scientific number is printed, so the data can be put directly on the screen. A/Y returns holding the address, so the string can also be output by using **$ab1e**, the sting printing routine.

7.3 Integer to FP

$bc44 = Convert signed 16-bit integer held in FAC1 at $63 (lo) and $62 (high) to FP
$b391 = Convert signed 16-bit integer held in Y/A (lo/high) to a FP number in FAC1
$bc3c = Convert unsigned 8-bit integer held in A to a FP number in FAC1
$b3a2 = Convert unsigned 8-bit integer held in Y to a FP number in FAC1

The first routine is the actual code. The other entries simply set up $62/$63 for you. This routine wrecks FAC2, so if it is needed, it should be loaded after the conversion. It also uselessly (from our ML standpoint) sets the data type flag at $0d. Skipping these routines and jumping straight to the normalization routine at **$b8d2** gives greater control, i.e. 24-bit and 32-bit integer inputs are possible, as are signed 8-bit inputs. For example, to convert a 24-bit number, load FAC1 with the number from the most-significant byte to the least. Store zero in the last mantissa byte, and also in the sign and rounding bytes ($66 and $70). Then set the exponent to 24 (24 + $80 = $98). Jump to $b8d2 with the carry indicating whether the result should be positive (set) or negative (clear).

The above process is basically what the KERNAL authors did to implement the 24 bit TI variable in BASIC (though they zero the high byte instead of the low, and hence use a mantissa of $a0).

You can take advantage of their work with the following pair of calls to convert a 24 bit unsigned integer held in YXA:

```
sec
jsr $af87  ; sets mantissa to 00yyxxaa
jsr $af7e  ; set rest of FAC1 and JMP to $b8d2
```

7.4 String to FP

$bcf3 = PETSCII string to FP in FAC1

With the exception of “E,” the PETSCII representation of a number does not differ from the screen code representation, so a number can be read directly off the screen. A character in the string not recognized as numeric (or +, -, E, and the decimal point) terminates the string. Spaces are ignored by this routine, so inputting “12 34” leaves “1234” in FAC1. If the exponent following E is longer than two digits, an overflow error occurs. Directly before jumping to this routine, store the address of the string in $7a/7b (low/high), then JSR $79, an entry into CHRGET, to properly process the first byte of the string.

Optimising

Multiply

Numerous advantages are gained improving the speed of the multiply. Routines that rely on it can be copied from the ROM and pointed to the new routine to improve performance. The following example relies on the Steve Judd's fast multiplication and reduces the number of cycles to multiply from around 2300 to little over 1400.

Fast Floating Point Multiply

Movement

These routines are called often, and move data around in memory using indirect indexing. The comparison also uses indirect indexing, and both can be rewritten to point to specific ZP locations instead. Frequently reused variables can be stored on the ZP for faster retrieval. Constants can be stored as an in-lined series of lda/sta instead of as a variable that must be loaded. Just ensure you mimic the sign comparison, and exit the loading of either FAC with FAC1's exponent in the accumulator.

Whole Number Exponents

In cases where you are simply squaring or cubing a number, simply multiplying them together is faster than relying on the exponent routines, which uses log and exp. Larger, whole-number exponents can also be calculated using exponentiation by squaring. For example:

$x^4 = (x^2)^2$
$x^8 = ((x^2)^2)^2$
$x^{16} = (((x^2)^2)^2)^2$

Using the additive law of exponents many combinations are possible, i.e. $x^{21} = x(x^4(x^{16}))$. From assembly, collecting the values needed for each term in sequence means that once you've calculated x^4, you're only two more multiplications away from the highest term. Four multiplications get you to x^{16}, followed by three more to multiply the collected terms. The result is seven multiplies. At this point

the built-in routine would be less than halfway done.

Faster Log and Exp

Logarithms can be calculated without expensive multiplies through a shift-and-add method. (Source: The "FXTbook" www.jjj.de/fxt) First a table needs to be generated. The number of terms in the table will determine the accuracy of the result. 20 iterations gets 7 digits of accuracy. For the table f(a) = $\log_n(1+1/2^a)$. In the BASIC example, the built-in log is $\log_e$.

```
5 dim t(20)
10 for a=1 to 20
15 t(a) = log(1+(1/(2^a)))
20 next
99 k=0:r=0:e=1:v=1:x=8
100 k=k+1:if k=20 goto 200
105 v=v*.5
110 u=e+e*v
115 if u>x goto 100
120 r=r+t(k)
125 e=u:goto 110
200 print r
205 print log(x)
```

The output is:

```
2.07944107
2.07944154
```

The assembly version, presuming the table is already made:

```
input   !byte $84,00,00,00,00 ;Floating Point 8
var_u   !byte $00,00,00,00,00
var_e   !byte $00,00,00,00,00
var_v   !byte $00,00,00,00,00
var_re  !byte $00,00,00,00,00
tempcount !byte 0


        lda #<$c000-5  ;assume table at $c000
    sta $fc
    lda #>$c000-5
    sta $fb

    lda #21
    sta tempcount
    lda #0
    sta var_re
    sta var_e+1
    sta var_e+2
```

```
    sta var_e+3
    sta var_e+4
    sta var_v+1
    sta var_v+2
    sta var_v+3
    sta var_v+4
    lda #$81
    sta var_e
    sta var_v

    ;the above loads e and v with 1, and re with 0

logloop dec tempcount
    beq logdone
    lda #5
    clc
    adc $fb
    sta $fb
    lda #0
    adc $fc
    sta $fc
    dec var_v    ;/2
inlogloop
    lda #<var_e
    ldy #>var_e
    jsr VARtoFAC
    jsr FACtoARG
    sec        ;returns with exponent
    sbc var_v    ;get difference
    sta $fd        ;temp
    lda $61        ;FAC1 exp
    sec
    sbc $fd
    sta $61        ;(e*v) division that is always a power of two
            ;a.k.a. exponent subtraction
    jsr addf    ;e+(e*v)
    ldx #<var_u
    ldy #>var_u
    jsr FACtoVAR
    lda #<input
    ldy #>input
    jsr FACcmp
    bne logloop
    lda $fb
    ldy $fc
    jsr VARtoFAC
    lda #<var_re
    ldy #>var_re    ;r=r+t(x)
    jsr add
    ldx #<var_re
```

```
    ldy #>var_re
    jsr FACtoVAR
    lda var_u+0    ;e=u
    sta var_e+0
    lda var_u+1
    sta var_e+1
    lda var_u+2
    sta var_e+2
    lda var_u+3
    sta var_e+3
    lda var_u+4
    sta var_e+4
    jmp inlogloop
logdone
    rts
```

This executes in around 13000 cycles, where the built-in routine takes 19000. Putting var_e and var_u on the ZP is worthwhile. Because we are only using the addition routines, a good amount of the ZP is opened up.

The shift-and-add Exp algorithm uses the same table as our new Log routine.

```
299 k=0:r=0:e=1:v=1:x=8
300 k=k+1:if k=20 goto 400
305 v=v*.5
310 u=r+t(k)
315 if u>x goto 300
320 r=u
325 e=e+e*v:goto 310
400 print e
405 print exp(x)
```

Output:

```
2980.95568
2980.95799
```

Doing log(exp(8)) = 7.99999624

If we do exponentiation with our new routines like the Kernal does (y^x = exp(x*log(y))) for 8^8 we get 16777148.6, while the true value is 16777224.

Faster Square Root

The standard square root, puts .5 into the exponent and drops into the exponentiation routine. This quicker (and more accurate) version was published in *Transactor* Issue 8.1, and was written by E.J. Schmahl. Rather than a series, it uses Newton's Method to fine tune a first guess based on the exponent. Two temporary FP areas are used. The root is returned in FAC1 and in the result variable. The routine works directly on the variables, so a small speed increase results if these are located on

the ZP.

```
sqrt    ldx #<value
    ldy #>value
    jsr FACtoVAR
    lda value+1
    bmi error    ;no negative square root
    lda value
    beq done    ;root of 0 is 0

    ldy #$00    ;fill temp result with 0
    sty result+1
    sty result+2
    sty result+3
    sty result+4

    lda value    ;get guess based on argument
        clc
    ror
    bcs sqrtadd
    ldx #$80
    stx result+1
sqrtadd adc #$40
    sta result
    lda value+1
    ora result+1
    lsr
    lsr
    lsr
    lsr
    tax
    lda sqrttable,x
    sta result+1
    lda #04        ;4 iterations of newton's method
    sta $fb
    lda #<result
    ldy #>result
    jsr VARtoFAC
-   lda #<value
    ldy #>value
    jsr div
    lda #<result
    ldy #>result
    jsr add
    dec $61
    ldx #<result
    ldy #>result
    jsr FACtoVAR
    dec $fb
    bne -
```

```
done    rts

error   rts

sqrttable
    !byte 03,11,18,25,32,38,44,50
    !byte 58,69,79,89,98,107,115,123
```

Error Handling

An error in BASIC terminates the program, which is undesirable when working with ML. When an error occurs, any number of values could have been pushed onto the stack. For ML, this means successful recovery depends on saving the stack pointer before jumping into routines that might generate an error. BASIC deals with this by having the error handling routine directly before the main loop. It resets the stack pointer to $fa, and falls through to print the READY prompt. Errors can be intercepted by changing the error handling vector at $300. The X register will hold the error number on entry. BASIC doubles this value and uses a look-up table to print the appropriate error message.

Address Table

Further Reading

Much if this information was sourced from *COMPUTE!'s VIC-20 and Commodore 64 Tool Kit: BASIC*, which basically is the comments for the disassembly minus the actual disassembly (copyright reasons). The algorithms used for the FP routines are described in great detail.