The Fast Floating-Point Library

By
Andrew Mui
axm2@cornell.edu

Autonomous Walking Robots Team
School of Engineering
Cornell University

May 21, 2007

Faculty Advisor: Professor Andy Ruina,
Theoretical & Applied Mechanics

Credit Option: 3 Credits for Fall 2006
Also for fulfillment of technical writing requirement
Acknowledgements

I would like to thank Professor Andy Ruina for having me as a member of the Autonomous Walking Robots Team. In one semester, I was given the opportunity to test motors, write Matlab programs to analyze the measured data, develop programs in assembly language, work with a microcontroller, and be a part of history by helping to create a robot that set a world record for greatest distance walked. I would also like to thank Jason Cortell for helping me develop the FFloat library, as well as helping me learn assembly programming. Finally, I would like to thank Daniel Karssen for all that he has contributed to the success of the Cornell Ranger. Though he was only in our lab for three months (he is a graduate student from the Netherlands), we would not have been able to create a record-breaking robot without his help.
0 Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/13/2006</td>
<td>Finished first final copy of report</td>
</tr>
<tr>
<td>5/14/2007</td>
<td>Added U32int2FFloat function</td>
</tr>
</tbody>
</table>
1 Introduction

The goal of our project was to create an efficient (low-power), robust, versatile, and autonomous walking quadruped (four-legged) robot. A walking robot that mimics the human gait has important applications in the field of medicine. For example, if a walking robot can be designed, a “wheelchair” that has legs instead of wheels can be designed. Such a device would enable the handicapped to travel up and down stairs, eliminating the need for wheelchair ramps. Someday, we might be able to replace prosthetics with powered robotic legs for people who have lost the ability to walk.

The most efficient robots, also known as passive-dynamic robots, walk by mimicking the human gait, and use gravity to travel down a slope. However, they cannot walk on flat ground, and will topple even if the surface beneath them is only slightly uneven, making them neither versatile nor robust. Our robot, the Cornell Ranger, uses a microcontroller to monitor and guide its motions. It has an outer pair and an inner pair of legs, and each leg has a movable ankle, whose motion is controlled by foot sensors. The robot walks by lowering the ankles on its front pair of legs when its front feet are near the ground, pushing off the ground with the heels of its front feet, and using its own momentum to swing the back pair of legs forward. Whenever the back pair of legs is swinging forward, the ankles are raised to prevent the feet from dragging on the ground.

The microcontroller monitors each part of the robot’s walk, and provides extra power to the ankles during push-off of the foot. This enables the robot to walk on flat ground (and even uphill on shallow slopes), and increases the robot’s stability as well (even though the feet on our robot were less than an inch wide, our robot was able to walk on an indoor track, which does not have a smooth surface). But since the microcontroller needs to be powered, our robot uses more power than gravity-based passive-dynamic robots.

Our microcontroller is designed to execute a list-of commands once every millisecond. These commands include determining hip and feet angles, and position of the feet relative to the ground, among other things. Therefore, our Digital Signal Processing (DSP) unit, which handles all of the calculations, must be able to quickly and accurately handle large ranges of numbers with relatively high precision. In addition, basic arithmetic operations, such as addition, subtraction, multiplication, division, and comparisons, must be done quickly.

This is where the FFloat (fast floating-point) numbers come in. This report will describe what FFloat numbers are, why we used them in the software of our robot, and how we developed the FFloat library of functions. A brief description of each of the functions in the FFloat library will then be given along with runtime (cycle count) data for all of the FFloat functions. Appendix A presents an overview of two’s complement and Appendix B contains the source code for the FFloat functions the author developed. Hopefully, the functions we wrote for the FFloat library, as well as the methods we used to develop the library, will be useful in future research and development where efficient and accurate data processing is critical.

---

1 The MC56F8347, made by Freescale Semiconductor, is a 16-bit hybrid controller with an integrated 16-bit DSP chip (the DSP56800E) [1].
2 The DSP56800E, also made by Freescale Semiconductor. It is a low-power, fast 16-bit DSP [2].
2 Why Use FFloat?

This section will attempt to motivate the decision to use FFloat numbers as the standard numerical type for performing calculations in the Cornell Ranger. The FFloat numerical type is a very non-intuitive, and non-standard way to represent numbers, but as we will see shortly, it was our best option.

There are two other numerical types that could have been used in place of FFloat: IEEE floating-point and fixed-point. IEEE floating-point is implemented into the C programming language as the float type. Fixed-point type uses scaled, or normalized, integers to represent fractional values.

The most important factor in deciding which numerical type to use was speed. Our microcontroller is designed to run its code indefinitely through a 1-millisecond loop, and our microprocessor only allows for 60000 cycles per loop. Therefore, our functions needed to be fast, requiring as few cycles as possible. With this in mind, we decided that IEEE-float was too slow for use in our microprocessor. For example, an arithmetic operation as simple as addition takes 250 cycles, which means that with IEEE-float, each loop will only be, at best, capable of performing 240 additions. IEEE-float also has many features that we did not need, such as error handling for NaN (not a number) and representation for positive and negative infinity. These cases would have to be handled in all IEEE floating-point functions, greatly increasing the number of cycles needed to run the functions.

Fixed-point notation, on the other hand, is fast because it uses integers instead of floating-point values. The problem with fixed-point, however, is that many digits are required in order to accurately represent numbers that require several decimal places. If too few digits are used, the precision of the resulting number will be too poor to use, and if too many digits are used, the number could overflow, giving clipped data, which would give a completely incorrect result.

In addition, fixed-point numbers must be scaled properly in order for fixed-point operations to work properly. Trying to figure out the scaling for every parameter during software development is time-consuming and prone to error. With floating-point numbers, the exponent automatically takes care of the scaling. This eliminates the need to manually scale floating-point numbers, reducing development time, as well as the time needed to fix bugs caused by incorrectly scaling numbers.

In short, FFloat numbers surpass IEEE floating-point in speed and fixed-point in precision and ease-of-use. Part of the reason that IEEE floating-point was so slow was that our DSP chip did not have a floating-point co-processor. For our next robot, we will be using a DSP chip with a floating-point co-processor, which will probably reduce code development time significantly.
3 An Overview of FFloat Numbers

3.1 Format for FFloat Numbers

FFloat numbers are 32-bit\(^3\) binary numbers. They are represented by a mantissa (a fractional value) multiplied by a base number raised to an exponent (an integral value).

The mantissa is a signed two’s complement\(^4\) 16-bit binary number, and is represented by the rightmost 16 bits of the FFloat number. It has a range between \(-2^{15}\) (-32768, 0x8000)\(^5\) and \(2^{15}-1\) (32767, 0x7FFF), but is interpreted as a fractional value between -1 (inclusive) and 1 (exclusive). To get the real fractional value, divide the FFloat mantissa value by \(2^{15}\).

The exponent is also a signed two’s complement 16-bit binary number, and is represented by the leftmost 16 bits of the FFloat number. Though we have 16 bits to work with, we will only be using the bottom 8 bits to represent the exponent value, which means that the range of the exponent is between \(-2^7\) (-128, 0xFF80) and \(2^7-1\) (127, 0x007F). The top 8 bits of the exponent are sign bits: all ones for negative values, and all zeros for nonnegative values.

To get the real floating-point numerical value, we take the real fractional value of the FFloat mantissa and multiply it by 2 raised to the exponent of the FFloat number:

\[
\text{numerical value} = \left(\frac{\text{FFloat mantissa}}{2^{15}}\right) \times 2^{\text{FFloat exponent}}.
\]

Even though we are only using 8 bits to represent the exponent, we can represent numbers between \(2^{-128}\) (2.93 \times 10^{-39}) and \(2^{127}\) (1.70 \times 10^{38}), which is more than sufficient for our calculations.

For convenience, we will represent FFloat numbers in hexadecimal rather than in binary, so that only 8 digits, instead of 32, are needed (a hexadecimal number can be represented by a 4-bit binary number).

3.2 Handling Exceptional Values

For FFloat numbers, there are three types of values that must be handled in a special manner: zero, overflow values, and underflow values.

Zero: We have decided to set zero equal to 0xFF800000. The mantissa is zero, and the exponent is set to its most negative value (-\(2^7\)). It was determined that if zero is defined this way, our FFloat addition and subtraction functions could be written without special exception handling for adding or subtracting zero, thereby saving processing time.

Overflow: Occurs if the magnitude of an FFloat number is larger than 0x007F7FFF (for positive values) or 0x007F8000 (for negative values). To check for overflow, the exponent of the FFloat number is compared to the value 0x007F (\(2^7-1\))\(^6\). If the FFloat number is determined to be too

---

\(^3\) A bit is a 1 or 0.
\(^4\) See Appendix A (Section 6) for a more in-depth discussion on two’s complement.
\(^5\) By convention, hexadecimal numbers are preceded by the characters 0x.
\(^6\) We used the built-in assembly instruction cmp.w, which compares two 16-bit words.
large in magnitude, the sign of the overflow (positive or negative) is then determined by comparing the FFloat number’s mantissa value to the mantissa values 0x7FFF and 0x8000 \(^7\). The overflow number is then set to the maximum positive or negative number as appropriate.

Underflow: Occurs if the magnitude of an FFloat number is smaller than 0xFF804000 (for positive values) or 0xFF80BFFF (for negative values). To check for underflow, the exponent of the FFloat number is compared to the value 0xFF80 (-2\(^7\)). If the FFloat number is determined to be too small in magnitude, the sign of the overflow (positive or negative) is then determined by comparing the FFloat number’s mantissa value to the mantissa values 0x4000 and 0xBFFF. The underflow number is then set to the minimum positive or negative number as appropriate.

Overflow range: less than 0x007F8000 (-2\(^{127}\)) or greater than 0x007F7FFF (.99997 \(\times 2^{127}\))
Underflow range: between 0xFF80BFFF (-.50003 \(\times 2^{-128}\)) and 0xFF804000 (.5 \(\times 2^{-128}\))

In our FFloat library functions, the Zero case is handled before the Underflow case so that the underflow check does not throw out an input of zero.

3.3 Using FFloat Numbers

3.3.1 Declaring and Initializing FFloat numbers in C

Since the ffloat\(^8\) type is not defined in C, any program that uses ffloat numbers must include the following type definition statement:

```c
typedef long unsigned int ffloat;
```

There are two ways to initialize an FFloat number. The first is to initialize it directly, by setting the variable equal to a hexadecimal number. For example, to initialize the ffloat variable ffnumA to zero, use the following statement:

```c
ffloat ffnumA = 0xFF800000;
```

The second way is to use one of the two integer-to-FFloat conversion functions, both of which are part of the FFloat library. To initialize ffnumA to zero using the functions, use the following statement:

```c
ffloat ffnumA = S16int2FFloat(0);
```

S16int2FFloat and S32int2FFloat are the two integer-to-FFloat conversion functions, and IEEE2FFloat is the standard float-to-FFloat conversion function. They will be discussed in greater detail in Section 4, along with the other FFloat library functions.

\(^7\) We used the built-in assembly instruction tst.w, which compares a 16-bit word to zero.

\(^8\) ffloat is the type definition for FFloat that was used in our C code.
3.3.2 Performing Operations on FFloat Numbers

Since FFloat numbers are not standard numerical types in C, there are no operators in C defined for FFloat numbers. Therefore, you must replace the arithmetic operators (+, -, \times, \div) and the comparison operators (\geq, \leq, =) with the appropriate function in the FFloat library.

Use of the trigonometric, conversion, and absolute value functions is similar to that in IEEE floating-point.

4 The FFloat Library

4.1 Development of the FFloat Library

After deciding to use FFloat numbers, we had to decide which functions we needed for the FFloat library. A disadvantage of FFloat numbers is that they are not a standard type in C. Therefore, we had to write functions for the basic arithmetic operators (+, -, \times, \div) and comparison operators (\geq, \leq, =), and use these functions in place of the operators. The trigonometric functions were developed because we needed a way to determine hip and foot angles in order to detect, for example, when the foot was about to touch the ground. Most of the other functions, such as taking the negative of a number and finding the absolute value of a number, were developed for use in the trigonometric functions and in the main C code.

In developing the FFloat library, we used CodeWarrior, a product of Metrowerks. We chose CodeWarrior for two main reasons: 1) it allowed us to write assembly code and C code in the same file (functions written in assembly code always start with the keyword asm), and 2) it supports the DSP56800, the DSP chip used in our microcontroller. As an added feature, CodeWarrior has a built-in DSP56800 simulator, which allowed us to measure the number of cycles and instructions required by the DSP56800 to execute a specific block of code (we designated this block of code using breakpoints).

The S16int2FFloat, S32int2FFloat, and IEEE2FFloat conversion functions (see Section 4.2) were the first to be developed. These functions were verified by calculating the expected FFloat value by hand, then printing out the output of the function to see if it matched our prediction. Once these three functions were written, checking the output of all the other FFloat functions was easy. As an example, here is how FFadd would have been verified:

1. Define ffloatA = S16int2FFloat(a) and ffloatB = S16int2FFloat(b), where a and b are 16-bit integers (short int). S32int2FFloat or IEEE2FFloat could also be used, based on the range of numbers that needed to be added.
2. Run FFadd(ffloatA, ffloatB) and print out the output.
3. Print out the value S16int2FFloat(a+b) and see if it matched the output given by FFadd.
4. Test all possible branches of FFadd, including exception cases such as overflow.

To reduce the cycle count, we wrote all of our functions in assembly (the trigonometric functions were written in C, however, because they were too complex to program in assembly).

Since we spent less than a month working on the FFloat library, we did not have time to optimize the assembly code that we wrote. However, our cycle count data will be useful in determining
which functions need to be optimized, and optimization of our functions in the future, when we have more time, is something that should be considered.

### 4.2 FFloat Library Functions

This section lists all the functions in the FFloat library by their function headers, and gives a brief description of what the functions return.

**Standard arithmetic operations**

- **asm ffloat FFadd(register ffloat ffnum1, register ffloat ffnum2)** – returns ffnum1+ffnum2.
- **asm ffloat FFsub(register ffloat ffnum1, register ffloat ffnum2)** – returns ffnum1-ffnum2.
- **asm ffloat FFmult(register ffloat ffnum1, register ffloat ffnum2)** – returns ffnum1×ffnum2.
- **asm ffloat FFdiv(register ffloat ffnum1, register ffloat ffnum2)** – returns ffnum1/ffnum2.

**Trigonometric functions**

- **ffloat FFsin(ffloat xin)** – returns the sine of xin.
- **ffloat FFcos(ffloat xin)** – returns the cosine of xin.
- **ffloat FFatan(ffloat xin)** – returns the arctangent of xin.

**Comparison functions**

- **asm bool FFgt(register ffloat ffnum1, register ffloat ffnum2)** – returns true iff ffnum1>ffnum2.
- **asm bool FFgte(register ffloat ffnum1, register ffloat ffnum2)** – returns true iff ffnum1≥ffnum2.
- **asm bool FFgtz(register ffloat ffnum)** – returns true iff ffnum>0.
- **asm bool FFlt(register ffloat ffnum1, register ffloat ffnum2)** – returns true iff ffnum1<ffnum2.
- **asm bool FFlte(register ffloat ffnum1, register ffloat ffnum2)** – returns true iff ffnum1≤ffnum2.
- **asm bool FFltz(register ffloat ffnum)** – returns true iff ffnum<0.
- **asm bool FFeqz(register ffloat ffnum)** – returns true iff ffnum=0.

**Conversion functions**

- **asm ffloat S16int2FFloat(register short int inum)** – returns the ffloat equivalent of inum.
- **asm short int FFloatRnd2S16int(register ffloat ffnum)** – returns the int16 equivalent of ffnum rounded up to the nearest integer.
- **asm short int FFloatTrunc2S16int(register ffloat ffnum)** – returns the int16 equivalent of ffnum truncated (fractional part of ffnum cut out).
- **asm ffloat S32int2FFloat(register long int inum)** – returns the ffloat equivalent of inum.
- **asm ffloat U32int2FFloat(register long unsigned int unum)** – returns ffloat equivalent of unum.
- **asm ffloat IEEE2FFloat(register float fnum)** – returns the ffloat equivalent of fnum.
- **float FFloat2IEEE(ffloat ffnum)** – returns the IEEE floating-point equivalent of ffnum.

**Other FFloat functions**

- **asm ffloat FFneg(register ffloat ffnum)** – returns the negative of ffnum.
- **asm ffloat FFabs(register ffloat ffnum)** – returns the absolute value of ffnum.

---

9 In C, bool is not a standard type. We defined it as an unsigned char with 1 being true and 0 being false.
4.3 Cycle Count Data for FFloat Library Functions

This section lists all the functions in the FFloat Library, along with their runtimes (measured in clock cycles). Worst-case values are used (i.e. the maximum number of cycles needed to execute a function). For comparison, we calculated the runtimes for the equivalent IEEE floating-point and integer operations as well (i.e. for FFadd, we calculated the runtime for a+b twice, once when a and b were IEEE floating-point numbers, and once when a and b were integers).

The trigonometric functions (FFsin, FFcos, and FFatan) were written in C, and called other FFloat functions, so cycle counts for those functions is much higher than the cycle counts for the other functions (which were written in assembly). In addition, the FFloat2IEEE function was not tested, because it was never used in the main code (though it was used frequently during development so that we could quickly and accurately calculate the FFloat values for all our parameters).

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Cycle Count</th>
<th>float cycle count</th>
<th>int cycle count</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFadd</td>
<td>53</td>
<td>254</td>
<td>6</td>
</tr>
<tr>
<td>FFsub</td>
<td>64</td>
<td>264</td>
<td>6</td>
</tr>
<tr>
<td>FFmult</td>
<td>41</td>
<td>230</td>
<td>8</td>
</tr>
<tr>
<td>FFdiv</td>
<td>87</td>
<td>335</td>
<td>40</td>
</tr>
<tr>
<td>FFsin</td>
<td>1061</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFcos</td>
<td>1050</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFatan</td>
<td>808</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFgt</td>
<td>46</td>
<td>157</td>
<td>18</td>
</tr>
<tr>
<td>FFgte</td>
<td>46</td>
<td>141</td>
<td>12</td>
</tr>
<tr>
<td>FFgtz</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFlt</td>
<td>46</td>
<td>164</td>
<td>12</td>
</tr>
<tr>
<td>FFlte</td>
<td>46</td>
<td>148</td>
<td>12</td>
</tr>
<tr>
<td>FFltz</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFeqz</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S16int2FFloat</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFFloatRnd2S16int</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFFloatTrunc2S16int</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S32int2FFloat</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U32int2FFloat</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IEEE2FFloat</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFneg</td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFabs</td>
<td>35</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1 Cycle Count data for FFloat library functions

The other FFloat functions are not supported by IEEE floating-point or integers in C (though the trigonometric functions are supported by IEEE floating-point numbers in C++). As shown from Table 1 above, FFloat is, on average, five times faster than IEEE floating-point, and four times slower than integers. Therefore, int operations are the fastest; however, the benefits gained from their speed is offset by the trouble needed to scale them properly.
5 Summary and Conclusion

We chose to use the FFloat (fast floating-point) numbers in the Autonomous Walking Robots project to achieve high data processing speed, wide dynamic range, and ease of software development. FFloat numbers allow us to complete a multiply operation in 41 cycles, making FFloat numbers more than ten times faster than IEEE floating-point numbers on our microprocessor. In addition, they give us an incredible 77 orders of magnitude to work with\textsuperscript{10}. At the same time, FFloat numbers do not need to be scaled – the exponent part of the number takes care of scaling.

In the end, the Cornell Ranger managed to walk 1003 meters (just over a kilometer) on an indoor track unassisted (with the exception of occasional steering done remotely to prevent it from hitting the building walls. As far as we know, this is a world record for farthest distance a robot has walked on its own. The success of the Cornell Ranger could not have been realized without the use of a fast and accurate number type like FFloat.

Next semester, our team will be designing a biped (two-legged) robot. For that robot, we will be using a DSP chip with a built-in high-speed floating-point coprocessor. Therefore, we probably will not be using our FFloat library for our next robot. However, if the FFloat functions prove to be better than implementing floating-point functions on the new DSP chip, it would be worthwhile to devote more time and effort to improving the functionality and efficiency of the functions in the FFloat library.

6 Appendix A: Two’s Complement

Two’s Complement is a way of representing negative values in binary. Instead of representing a power of two, the top bit in a two’s complement binary number is a sign bit, with 0 representing a positive number and 1 representing a negative number. Two’s complement is used because it handles overflow errors without the need for special exception cases, decreasing processing times.

To find the negative of a number, we must invert all the bits in that number and add one to the result [3]. For example, starting from the 8-bit representation of 21,

\[0001 \ 0101,\]

you can get the 8-bit representation of -21 by first inverting all the bits, as shown:

\[1110 \ 1010,\]

then adding one to the result. The 8-bit representation of -21 would look like this:

\[1110 \ 1011.\]

Since this process finds the two’s complement (negative) of a binary number, this process works in reverse as well (going from negative to positive numbers).

\textsuperscript{10} Exponent can represent values between \(2^{-128} \ (2.93 \times 10^{-39})\) and \(2^{127} \ (1.70 \times 10^{38})\). Mantissa can represent values between \(-1\) and 1.
Appendix B: Code for FFloat library functions

This section provides source code for all the FFloat functions. I wrote most of the comparison functions (FFgt, FFgte, FFlt, FFlte, and FFeqz) as well as a couple of the conversion functions (S16int2FFloat and S32int2FFloat). The other functions were written by Jason Cortell.

7.1 FFadd

asm ffloat FFadd(register ffloat ffnum1, register ffloat ffnum2)
{
  move.w    A0,X0     //Store ffnum1 mantissa temporarily in X0
  move.w    B0,Y0     //Store ffnum2 mantissa temporarily in Y0
  move.w    A1,Y1     //Put ffnum1 exponent (exp1) in Y1
  sub        B,Y1     //Y1 = exp1 - exp2
  
  //Setup: Larger ffnum exponent goes in Y0; mantissa to be shifted goes in B1;
  //mantissa to stay the same goes in A1; abs exp difference goes in Y1
  tlt        B,A      //Move ffnum2 (mantissa and exp) to A (not
                     //shifted) if Y1 neg
  tlt        X0,B     //Move ffnum1 mantissa to B1 for shifting if Y1
                     //neg
  tge        Y0,B     //Move ffnum2 mantissa to B1 for shifting if Y1
                     //not negative
  abs        Y1       //positive shift values
  cmp.w      #15,Y1   //More than 15-bit shift (ASRAC only works to
                   //15 bits)?
  jgt        Neglect  //If yes, an input ffnum will go to zero if
                     //shifted
  move.w    A1,Y0     //Move larger exp to Y0 for shifting
  move.w    A0,A      //Move mantissa A0 to A1 for adding
  asrac      B1,Y1,A   //Extend B1 to 36 bits, shift right by
                     //Y1, and add to A
  asr        A        //Shift right to prevent overflow of CLB (next)
  clb        A,X0     //Count sign bits
  asll.l     X0,A     //Normalize
  tst.w      A1       //Check if relevant part of result is zero
  jeq        Zero     //Result is zero
  sub        X0,Y0    //Adjust exponent of expl
  inc.w      Y0       //Return to normal scale
  clb        Y0,X0    //Check number of sign bits in exponent
  cmp.w      #8,X0    //If less than 8 (exp > 8 bits),
  jlt        Exp_Err  //Jump to exponent exception handler

Continue:

11 This function was originally FFeq. It was later modified to become FFeqz, FFltz, and FFgtz.
rnd A //round to 16 bits in A1
rtsd //delayed return from subroutine
move.w A,A0 //Move mantissa of sum to lower word of ffnum1
//return value
move.w Y0,A1 //Move exponent to upper word of ffnum1 (return
//value)
sxt.l A //Sign-extend A to 36 bits
//end of main add function

Zero:
rtsd //Delayed return from subroutine - will execute
//next three words
move.w #$FF80,A //Set exp of sum to minimum
clr.w A0 //Set mantissa of sum to 0
//end of zero handler

Exp_Err:
    cmp.w #$007F,Y0  //If not overflow, go to underflow check
    jle Underflow
    tst.w A1  //Positive or negative overflow?
    jlt NegO  //If negative, go to negative handler
    move.w #$007F,A  //Max out exponent
    rtsd  //Delayed return from subroutine - will execute
    //next three words
    move.w #$7FFF,A0  //Max out mantissa
    nop  //Delay slot filler
//end

NegO:
    move.w #$007F,A  //Max out exponent
    rtsd  //Delayed return from subroutine - will execute
    //next three cycles
    move.w #$8000,A0  //Most negative mantissa
    nop  //Delay slot filler
//end

Underflow:
    cmp.w #$FF80,Y0  //Check for underflow
    jge Continue  //Not an error
    tst.w A1  //Positive or negative underflow?
    jlt NegU  //If negative, go to negative handler
    move.w #$FF80,A  //Minimum exponent
    rtsd  //Delayed return from subroutine - will execute
    //next three words
    move.w #$4000,A0  //Minimum normalized positive mantissa
    nop  //Filler for third delay slot
//end

NegU:
    move.w #$FF80,A  //Minimum exponent
    rtsd  //Delayed return from subroutine - will execute
    //next three words
    move.w #$BFFF,A0  //Minimum (abs) normalized negative mantissa
    nop  //Filler for third delay slot
//end of E_Err

Neglect:
    rts  //The input with the larger exp becomes the
    //output
}

7.2 FFsub
asm ffloat FFsub(register ffloat ffnum1,register ffloat ffnum2) {

move.w A0,X0   //Store ffnum1 mantissa temporarily in X0
move.w B1,Y1   //Store ffnum2 mantissa temporarily in Y1
move.w B0,B    //Prepare to negate B
asr B          //Prevent overflow
inc.w Y1       //Adjust exponent
neg B          //Negate
clb B,Y0       //Count leading bits
asll.l Y0,B    //Rescale
sub Y0,Y1      //Adjust exponent
move.w B1,Y0
move.w Y1,B
move.w Y0,B0
move.w A1,Y1   //Put ffnum1 exponent (exp1) in Y1
sub B,Y1       //Y1 = exp1 - exp2

//Setup: Larger ffnum exponent goes in Y0; mantissa to be shifted goes in B1; mantissa to stay the same goes in A1; abs exp difference goes in Y1

tlt B,A         //Move ffnum2 (mantissa and exp) to A (not shifted) if Y1 neg
tlt X0,B        //Move ffnum1 mantissa to B1 for shifting if Y1 neg
tge Y0,B        //Move ffnum2 mantissa to B1 for shifting if Y1 not negative
abs Y1          //Positive shift values
cmp.w #15,Y1    //More than 15-bit shift (ASRAC only works to 15 bits)?
jgt Neglect     //If yes, an input ffnum will go to zero if shifted
move.w A1,Y0    //Move larger exp to Y0 for shifting
move.w A0,A     //Move mantissa A0 to A1 for adding
asrac B1,Y1,A   //Extend B1 to 36 bits, shift right by Y1, and add to A
asr A           //Shift right to prevent overflow of CLB (next)
clb A,X0        //Count sign bits
asll.l X0,A     //Normalize
tst.w A1        //Check if relevant part of result is zero
jeq Zero        //Result is zero
sub X0,Y0       //Adjust exponent of exp1
inc.w Y0        //Return to normal scale
clb Y0,X0       //Check size of exponent word
cmp.w #8,X0     //Check size of exponent word
jlt Exp_Err
Continue:
    rnd A         //Round to 16 bits
    rtsd          //Delayed return from subroutine
move.w A,A0 //Move mantissa of sum to lower word of ffnum1
    // (return value)
move.w Y0,A1 //Move exponent to upper word of ffnum1 (return
    // value)
sxt.l A //Sign-extend A to 36 bits
  //end of main add function

Zero:
  rtsd //Delayed return from subroutine - will
         // execute next three inst.
  move.w #$FF80,A //Set exp of sum to minimum
  clr.w A0 //Set mantissa of sum to 0
  //end of zero handler

Exp_Err:
  cmp.w #$007F,Y0
  jle Underflow //If not overflow, go to underflow check
  tst.w A1 //Positive or negative overflow?
  jlt NegO //If negative, go to negative handler
  move.w #$007F,A //Max out exponent
  rtsd //Delayed return from subroutine - will execute
         // next three words
  move.w #$7FFF,A0 //Max out mantissa
  nop //filler for third delay slot
  //end

NegO:
  move.w #$007F,A //Max out exponent
  rtsd //Delayed return from subroutine - will
         // execute next three words
  move.w #$8000,A0 //Most negative mantissa
  nop //filler for third delay slot
  //end

Underflow:
  cmp.w #$FF80,Y0 //Check for underflow
  jge Continue //Not an error
  tst.w A1 //Positive or negative underflow?
  jlt NegU //If negative, go to negative handler
  move.w #$FF80,A //Minimum exponent
  rtsd //Delayed return from subroutine - will execute
         // next three inst.
  move.w #$4000,A0 //Minimum normalized positive mantissa
  nop //filler for third delay slot
  //end

NegU:
  move.w #$FF80,A //Minimum exponent
  rtsd //Delayed return from subroutine - will execute
         // next three inst.
  move.w #$BFFF,A0 //Minimum (abs) normalized negative mantissa
  nop //filler for third delay slot
  //end of E_Err

Neglect:
  rts //The input with the larger exp becomes the
       //output

7.3 FFmult
asm ffloat FFmult(register ffloat ffnum1, register ffloat ffnum2)
{
  move.w B1,Y1 //This is to save exp2, use B for mult, and
                 //prepare for exp add
move.w A0,X0  //Can't multiply A0,B0 directly
move.w B0,Y0
mpyr X0,Y0,B  //Multiply with round; result unlikely to
differ from mpy, since truncated later
asr B  //Shift right, so CLB can give correct count
clb B,X0  //Count sign bits for normalization
asll.l X0,B  //Normalize
tst.w B1  //Check if relevant part of result is zero
jeq Zero  //Go to zero handler
add A,Y1  //add A1 to Y1
sub X0,Y1  //Update exponent after normalization
inc.w X0,Y1  //Return to normal scale
clb Y1,Y0  //count sign bits in exponent word
cmp.w #8,Y0  //If <8 (exp > 8 bits),
jlt Exp_Err  //jump to exponent exception handler

Continue:
  rtsd  //return with 3-cyle delay
  move.w Y1,A  //Put exp in return register
  rnd B  //Round to 16 bits in B1
  move.w B1,A0  //Move mantissa to A0
  //end of mult routine

Zero:
  rtsd  //return with 3-cyle delay
  move.w #$FF80,A  //Set exp of sum to minimum
  clr.w A0  //Set mantissa of sum to 0
  //end of zero handler

Exp_Err:
  cmp.w #$007F,Y1  //Check for overflow
  jle Underflow  //If not overflow, go to underflow check
  tst.w B1  //Positive or negative overflow?
  jlt NegO  //If negative, go to negative handler
  move.w #$7FFF,A0  //Max out mantissa
  rtsd  //Delayed return - will execute next three
          //words
  nop  //Filler for third delay slot
  //end

NegO:
  move.w #$007F,A  //Max out exponent
  rtsd  //Delayed return - will execute next three
          //words
  move.w #$8000,A0  //Most negative mantissa
  nop  //Filler for third delay slot
  //end

Underflow:
  cmp.w #$FF80,Y1  //Check for underflow
  jge Continue  //Not an error - continue normal code
  tst.w B1  //Positive or negative overflow?
  jlt NegU  //If negative, go to negative handler
  move.w #$FF80,A  //Minimum exponent
  rtsd  //Delayed return - will execute next three
          //words
  move.w #$4000,A0  //Minimum normalized positive mantissa
  nop  //Filler for third delay slot
  //end

NegU:
  move.w #$FF80,A  //Minimum exponent
  rtsd  //Delayed return - will execute next three
7.4 **FFdiv**

```c
asm ffloat FFdiv(register ffloat ffnum1, register ffloat ffnum2) {
    move.w A1,X0 //Move exponent of ffnum1 to X0
    move.w B1,Y0 //Move exponent of ffnum2 to Y0
    move.w A0,Y1 //Move mantissa of ffnum1 to Y1 for sign check
    move.w A0,A //Move mantissa of ffnum1 to A1
    move.w B0,B //Move mantissa of ffnum2 to B1
    eor.w B,Y1 //Calculate sign of final result
        //(sign bit of result will be 1=negative if
        //inputs signs differ)
    abs A
    abs B
    jeq DivZero //ffnum2 cannot be zero

L1:
    cmp A,B //Check result of B - A
    bgt L2 //Ready to divide
    brad L1 //Recheck (delayed branch)
    asr A //Reduce ffnum1 mantissa by factor of 2
    inc.w X0 //Increase ffnum1 exponent by one
    //end

L2:
    //Division of Positive Fractional Data (A1:A0 / B1)
    BFCLR #$0001,SR //Clear carry bit: required for 1st DIV
    instruction
        //REP #16
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        DIV B1,A //Form positive quotient in A0
        move.w A0,A //Move A0 to A1

    tst.w Y1 //Check sign needed for final result
    BGE L3 //Branch if final sign is non-neg
    NEG A //Negate mantissa if result is neg

L3:
    clb A,Y1 //Count sign bits
```
asll.l Y1,A //Normalize
tst A //Check if relevant part of result is zero
jeq Zero //Result is zero
sub Y0,X0 //Adjust exponent of exp1
sub Y1,X0
clb X0,Y0 //check size of exponent word
cmp.w #8,Y0
jlt Exp_Err

Continue:
RTSD
MOVE.W A,A0
MOVE.W X0,A1
sxt.l A //Sign-extend A to 36 bits
//END

DivZero:
//Call error handler here
MOVE.W #$007F,A //Needs work here
RTSD
MOVE.W #$7FFF,A0
NOP
//END

Zero:
RTSD
MOVE.W #$FF80,A
CLR.W A0
//END

Exp_Err:
cmp.w #$007F,X0 //If not overflow, go to underflow check
jle Underflow //Positive or negative overflow?
jlt NegO //If negative, go to negative handler
move.w #$007F,A //Max out exponent
rtsd //Delayed return from subroutine - will
//execute next three words
move.w #$7FFF,A0 //Max out mantissa
nop //END

NegO:
move.w #$007F,A //Max out exponent
rtsd //Delayed return from subroutine - will
//execute next three words
move.w #$8000,A0 //Most negative mantissa
nop //filler for third delay slot
//END

Underflow:
cmp.w #$FF80,X0 //Check for underflow
jge Continue //Not an error
tst.w A1 //Positive or negative underflow?
jlt NegU //If negative, go to negative handler
move.w #$FF80,A //Minimum exponent
rtsd //Delayed return from subroutine - will
//execute next three words
move.w #$4000,A0 //Minimum normalized positive mantissa
nop //Filler for third delay slot
NegU:

```assembly
move.w #$FF80,A    //Minimum exponent
rtsd               //Delayed return from subroutine - will
                    //execute next three words
move.w #$BFFF,A0   //Minimum (abs) normalized negative mantissa
nop                 //filler for third delay slot

@end of E_Err
```

### 7.5 FFsin

```c
ffloat FFsin(ffloat xin)
{
    int k, klo, khi;
    ffloat xdiff0, xdiff1;
    ffloat x = xin;
    static ffloat xlo = 0x00029b78;
    static ffloat xhi = 0x00026487;
    static ffloat ya[31] = {0xffccb968, 0xfffe958c, 0xffff97e0, 0x0000b4c3,
                            0x00009126, 0x00008643, 0x000080b3, 0x000080b3, 0x00008643,
                            0x00009126, 0x00008643, 0x000080b3, 0x000080b3, 0x00008643,
                            0x00009126, 0x00008643, 0x000080b3, 0x000080b3, 0x00008643,
                            0x00009126, 0x00008643, 0x000080b3, 0x000080b3, 0x00008643};
    static ffloat y2a[31] = {0xff800000, 0xfffd6a0f, 0xfffe67be, 0xffff4af6,
                             0xffff5ec6, 0xfffe672, 0xffff794a, 0xffff7ed5, 0xffff7ed5, 0xffff794a,
                             0ffield6e72, 0ffield5ec6, 0ffield4af6, 0ffield67be, 0ffield60e0, 0xffff0000,
                             0ffield95f0, 0ffield9841, 0ffieldb509, 0ffielda139, 0ffield918d, 0ffield86b5,
                             0ffield812a, 0ffield812a, 0ffield86b5, 0ffield918d, 0ffielda139, 0ffieldb509,
                             0ffield9841, 0ffield95f0, 0ffield95f0};
    static int numpoints = 31;
    static ffloat h = 0xfffe6b3b;
    static ffloat hinv = 0x00034c64;
    static ffloat pi2=0x00036487;
    static ffloat pi2inv=0xffff517c;

    if(FFlt(xin,xlo)){
        x=FFadd(
            xin,
            FFmult(
                S16int2FFloat(
                    FFloatTrunc2S16int(
                        FFsub(xhi,xin),
                        pi2inv)
                    ),
                    pi2)
            );
    }else if(FFgt(xin,xhi)){
        x=FFsub(
            xin,
            FFmult(
```
}
7.6 **FFcos**

```c
float FFcos(float xin)
{
    int k, klo, khi;
    float xdiff0, xdiff1;
    float x = xin;
    static float xl0 = 0x00029b78;
    static float xhi = 0x00026487;
    static float ya[31] = {0x00008000, 0x000082cc, 0x00008b10, 0x00009872, 0x0000aa59, 0xffff8000, 0xffffb0e4, 0xfffd94f6, 0xfffd6b09, 0xffff4f1b, 0x00004000, 0x000055a6, 0x0000678d, 0x000074ef, 0x00007d33, 0x00014000, 0x00007d33, 0x000074ef, 0x0000678d, 0x000055a6, 0x00004000, 0xffff4f1b, 0xfffd6b09, 0xfffd94f6, 0xffffb0e4, 0xffff8000, 0x0000aa59, 0x00009872, 0x00008b10, 0x000082cc, 0x00008000};
    static float y2a[31] = {0xff800000, 0xffff7cbe, 0xffff7481, 0xfffff556, 0xffffe7f88, 0xffffe4ed1, 0xffffc6aa5, 0xffff955a, 0xffffe517c, 0xffffe7f88, 0xfffff556, 0xfffff7481, 0xfffff7cbe};
    static int numpoints = 31;
    static float h = 0xffffe6b3b;
    static float hinv = 0x00034c64;
    static float pi2 = 0x00036487;
    static float pi2inv = 0xfffe517c;

    if (FFlt(xin, xlo))
    {
        x = FFadd(
            xin,
            FFmult(
                S16int2FFloat(
                    FFfloatTrunc2S16int(
                        FFmult(
                            FFsub(xin, xlo),
                            pi2inv
                        ),
                        pi2
                    ),
                ),
                pi2
            );
    }

    klo = FFfloatTrunc2S16int(FFmult(FFsub(x, xlo), hinv));
    khi = klo + 1;
    xdiff0 = FFsub(x, FFadd(xlo, FFmult(h, S16int2FFloat(klo))));
    xdiff1 = FFsub(xdiff0, h);
    return ( FFadd(ya[klo], FFadd(FFmult(FFmult(FFsub(ya[khi], ya[klo]), hinv), xdiff0), FFmult(FFmult(y2a[khi], xdiff0), xdiff1))) )
};
```


7.7 \textit{FFatan} \\

\texttt{ffloat FFatan(ffloat xin)\
{\begin{verbatim}
        int k, klo, khi;
        ffloat xdiff0, xdiff1;
        ffloat x = xin;
        static ffloat xlo = 0x0005b000;
        static ffloat xhi = 0x00055000;
        static ffloat ya[151] = {0x00019eaa, 0x00019eb5, 0x00019ec0, 0x00019ecc, 0x00019ed8, 0x00019ee4, 0x00019ef1, 0x00019ef8, 0x00019f28, 0x00019f36, 0x00019f46, 0x00019f55, 0x00019f66, 0x00019f6e, 0x00019f88, 0x00019f99, 0x00019f9f, 0x00019fac, 0x00019fbf, 0x00019fd3, 0x00019fe8, 0x00019ff2, 0x00019ff8, 0x0001a013, 0x0001a02a, 0x0001a042, 0x0001a05b, 0x0001a075, 0x0001a090, 0x0001a0ac, 0x0001a0c5, 0x0001a109, 0x0001a12b, 0x0001a14e, 0x0001a173, 0x0001a19a, 0x0001a1b7, 0x0001a1e6, 0x0001a21c, 0x0001a24c, 0x0001a27f, 0x0001a2b5, 0x0001a2ef, 0x0001a32c, 0x0001a36d, 0x0001a3bd, 0x0001a3fd, 0x0001a44d, 0x0001a4a2, 0x0001a4ff, 0x0001a563, 0x0001a5d0, 0x0001a646, 0x0001a6c7, 0x0001a754, 0x0001a7f0, 0x0001a89d, 0x0001a95d, 0x0001aa33, 0x0001ab25, 0x0001ac37, 0x0001ad71, 0x0001aed0, 0x0001b07f, 0x0001b26e, 0x0001b4bc, 0x0001b785, 0x0001bf81, 0x0001bf38, 0x0001bf94, 0x0001b957, 0x00009a2, 0xffff8292, 0xffffbd49, 0xff800000, 0xffff42b6, 0xffff76d, 0x0000565d, 0x000068a8, 0x000076b1, 0x000140c7, 0x0001450e, 0x0001487a, 0x00014b43, 0x00014d91, 0x00014f80, 0x00015125, 0x000153c8, 0x000154a, 0x000155cc, 0x000156a2, 0x00015762, 0x0001580f, 0x000159ab, 0x00015938, 0x0001599b, 0x00015a2f, 0x00015a9c, 0x00015b00, 0x00015bb2, 0x00015c02, 0x00015c4c, 0x00015c92, 0x00015cd3, 0x00015d10, 0x00015d4a, 0x00015d80, 0x00015db3, 0x00015de3,
\end{verbatim}}

)}
static ffloat y2a[151] = {0xff800000, 0xfff443e4, 0xfff446b6, 0xfff449b0, 0xfff44cd5, 0xfff45029, 0xfff453af, 0xfff4576a, 0xfff45b5f, 0xfff45f92, 0xfff46408, 0xfff468c6, 0xfff46dd1, 0xfff47331, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xfff47f0a, 0xfff478ec, 0xFFFgt

//return true if ffnum1>ffnum2, false otherwise
asm bool FFgt(register ffloat ffnum1, register ffloat ffnum2) {
    //First compare signs of numbers
}
tst.w A0
blt CheckSignANeg

//a is nonnegative
tst.w B0
//Both numbers are nonnegative - nonnegative exponents case
bge CasePNumExp
//If b is negative, a>b
rtsd
move.w #1,Y0
nop
nop

//a is negative
CheckSignANeg:
tst.w B0
//Both numbers are negative - negative exponents case
blt CaseNNumExp
//If b is nonnegative, a<b
rtsd
move.w #0,Y0
nop
nop

//If a and b are positive, go here
//larger exponent = larger #
CasePNumExp:
//move exponent data to X0 and Y0 registers for comparison
move.w A1,X0
move.w B1,Y0
cmp.w X0,Y0
blt aGTb //if(expB<expA) then a>b
bgt aNotGTb //if(expB>expA) then !(a>b)

//If exponents are equal, check mantissas
move.w A0,X0
move.w B0,Y0
cmp.w X0,Y0
blt aGTb //if(mantissaB<mantissaA) then a>b
rtsd
move.w #0,Y0
nop
nop

//If a and b are negative, go here
//larger exponent = smaller #
CaseNNumExp:
//move exponent data to X0 and Y0 registers for comparison
move.w A1,X0
move.w B1,Y0
cmp.w X0,Y0
bgt aGTb //if(expB>expA) then a>b
blt aNotGTb //if(expB<expA) then !(a>b)

//If exponents are equal, check mantissas
move.w A0,X0
move.w B0,Y0
cmp.w X0,Y0
//if (mantissaB < mantissaA) then a > b
blt aGTb
rtsd
move.w #0, Y0
nop
nop

//if a > b, go here
aGTb:
  rtsd
  move.w #1, Y0
  nop
  nop

//if a <= b, go here
aNotGTb:
  rtsd
  move.w #0, Y0
  nop
  nop

7.9 FFgte
//return true if a >= b, false otherwise
asm bool FFgte(register ffloat a, register ffloat b)
{
  //First compare signs of numbers
  tst.w A0
  blt CheckSignANeg

  //a is nonnegative
  tst.w B0
  bge CasePNumExp

  //If b is negative, a >= b
  rtsd
  move.w #1, Y0
  nop
  nop

  //a is negative
  CheckSignANeg:
    tst.w B0
    bge CaseNNumExp

    //If b is nonnegative, a < b
    rtsd
    move.w #0, Y0
    nop
    nop

  //If a and b are positive, go here
  //larger exponent = larger #
  CasePNumExp:
    //move exponent data to X0 and Y0 registers for comparison
    move.w A1, X0
    move.w B1, Y0
cmp.w X0, Y0
blt aGTEb  // if(expB<expA) then a>=b
bgt aNotGTEb // if(expB>expA) then !(a>=b)

// If exponents are equal, check mantissas
move.w A0, X0
move.w B0, Y0
cmp.w X0, Y0
ble aGTEb  // if(mantissaB<=mantissaA) then a>=b
rtsd
move.w #0, Y0
nop
nop

// If a and b are negative, go here
// larger exponent = smaller #
CaseNNumExp:
  // move exponent data to X0 and Y0 registers for comparison
  move.w A1, X0
  move.w B1, Y0
cmp.w X0, Y0
  bgt aGTEb  // if(expB>expA) then a>b
  blt aNotGTEb // if(expB<expA) then !(a>b)
  // If exponents are equal, check mantissas
  move.w A0, X0
  move.w B0, Y0
cmp.w X0, Y0
  ble aGTEb  // if(mantissaB<=mantissaA) then a>=b
  rtsd
  move.w #0, Y0
  nop
  nop

// if a>=b, go here
aGTEb:
  rtsd
  move.w #1, Y0
  nop
  nop

// if a<b, go here
aNotGTEb:
  rtsd
  move.w #0, Y0
  nop
  nop
}

7.10 FFgtz
asm bool FFgtz(register ffloat ffnum)
{
  // Test ffnum mantissa
tst.w A0
  bgt Positive
  // ffnum <= 0
rtsd //delayed return
clr.w Y0 //return value 0
nop //first filler instruction
nop //second filler instruction
//end

Positive:
//ffnum > 0
rtsd //delayed return
move.w #1,Y0 //return value 1
nop //first filler instruction
nop //second filler instruction
//end

Negative:
//ffnum < 0
rtsd //delayed return
move.w #1,Y0 //return value 1
nop //first filler instruction
nop //second filler instruction
//end

7.11 FFlt
//return true if ffnum1<ffnum2, false otherwise
asm bool FFlt(register ffloat ffnum1, register ffloat ffnum2)
{
    //First compare signs of numbers
tst.w A0
blt CheckSignANeg

    //a is nonnegative
tst.w B0
    //Both numbers are nonnegative - nonnegative exponents case
bge CasePNumExp
    //If b is negative, !(a<b)
    rtsd
    move.w #0,Y0
    nop
    nop

    //a is negative
CheckSignANeg:
  tst.w B0  //Both numbers are negative - negative exponents case
  blt    CaseNNumExp  //If b is nonnegative, a<b
  rtsd
  move.w #1,Y0
  nop
  nop

  //If a and b are positive, go here
  //larger exponent = larger #
  CasePNumExp:
              //move exponent data to X0 and Y0 registers for comparison
    move.w A1,X0
    move.w B1,Y0
    cmp.w X0,Y0
    bgt    aLTb  //if(expB>expA) then a<b
    blt    aNotLTb //if(expB<expA) then !(a<b)

              //If exponents are equal, check mantissas
    move.w A0,X0
    move.w B0,Y0
    cmp.w X0,Y0
    bgt    aLTb  //if(mantissaB>mantissaA) then a<b
    rtsd
    move.w #0,Y0
    nop
    nop

  //If a and b are negative, go here
  //larger exponent = smaller #
  CaseNNumExp:
              //move exponent data to X0 and Y0 registers for comparison
    move.w A1,X0
    move.w B1,Y0
    cmp.w X0,Y0
    blt    aLTb  //if(expB<expA) then a<b
    bgt    aNotLTb //if(expB>expA) then !(a<b)

              //If exponents are equal, check mantissas
    move.w A0,X0
    move.w B0,Y0
    cmp.w X0,Y0
    bgt    aLTb  //if(mantissaB>mantissaA) then a<b
    rtsd
    move.w #0,Y0
    nop
    nop

  //if a<b, go here
  aLTb:
  rtsd
  move.w #1,Y0
  nop
  nop

  //if a>=b, go here
aNotLTb:
    rtsd
    move.w   #0,Y0
    nop
    nop

7.12 FFlte
//return true if a<=b, false otherwise
asm bool FFlte(register ffloat a, register ffloat b)
{
    //First compare signs of numbers
    tst.w A0
    blt       CheckSignANeg

    //a is nonnegative
    tst.w B0
    //Both numbers are nonnegative - nonnegative exponents case
    bge       CasePNumExp
    //If b is negative, !(a<=b)
    rtsd
    move.w   #0,Y0
    nop
    nop

    //a is negative
    CheckSignANeg:
    tst.w B0
    //Both numbers are negative - negative exponents case
    blt       CaseNNumExp
    //If b is nonnegative, a<b
    rtsd
    move.w   #1,Y0
    nop
    nop

    //If a and b are positive, go here
    //larger exponent = larger #
    CasePNumExp:
    //move exponent data to X0 and Y0 registers for comparison
    move.w   A1,X0
    move.w   B1,Y0
    cmp.w    X0,Y0
    bgt       aLTEb       //if(expB>expA) then a<=b
    blt       aNotLTEb    //if(expB>expA) then !(a<=b)

    //If exponents are equal, check mantissas
    move.w   A0,X0
    move.w   B0,Y0
    cmp.w    X0,Y0
    bge      aLTEb       //if(mantissaB>=mantissaA) then a>=b
    rtsd
    move.w   #0,Y0
    nop
    nop
// If a and b are negative, go here
// larger exponent = smaller #
CaseNNumExp:
    // move exponent data to X0 and Y0 registers for comparison
    move.w A1,X0
    move.w B1,Y0
    cmp.w X0,Y0
    blt  aLTEb  // if(expB<expA) then a<=b
    bgt  aNotLTEb // if(expB>expA) then !(a<=b)

    // If exponents are equal, check mantissas
    move.w A0,X0
    move.w B0,Y0
    cmp.w X0,Y0
    bge aLTEb  // if(mantissaB>=mantissaA) then a>=b
    rtsd
    move.w #0,Y0
    nop
    nop

    // if a<=b, go here
aLTEb:
    rtsd
    move.w #1,Y0
    nop
    nop

    // if a>b, go here
aNotLTEb:
    rtsd
    move.w #0,Y0
    nop
    nop
}

7.13 FFltz
asm bool FFltz(register ffloat ffnum)
{
    // Test ffnum mantissa
    tst.w A0
    blt Negative

    // ffnum >= 0
    rtsd  // delayed return
    clr.w Y0  // return value 0
    nop  // first filler instruction
    nop  // second filler instruction
    // end

Negative:
    // ffnum < 0
    rtsd  // delayed return
    move.w #1,Y0  // return value 1
    nop  // first filler instruction
    nop  // second filler instruction
    // end
}
asm bool FFeqz(register ffloat ffnum)
{
    //Test ffnum mantissa
    tst.w   A0
    beq    Zero

    //ffnum != 0
    rtsd    //delayed return
    clr.w  Y0    //return value 0
    nop    //first filler instruction
    nop    //second filler instruction
    //end

Zero:
    //ffnum < 0
    rtsd    //delayed return
    move.w #1,Y0    //return value 1
    nop    //first filler instruction
    nop    //second filler instruction
    //end
}

7.14 FFeqz
//return true if ffnum=0, false otherwise
asm bool FFeqz(register ffloat ffnum)
{
    //Test ffnum mantissa
    tst.w   A0
    beq    Zero

    //ffnum != 0
    rtsd
    clr.w  Y0
    nop
    nop

Zero:
    //ffnum < 0
    rtsd
    move.w #1,Y0    //return value 1
    nop
    nop
}

7.15 S16int2FFloat
//convert an int16 to an ffloat value
asm ffloat S16int2FFloat(register short int inum)
{
    tst.w Y0
    jeq    Zero

    //inum != 0
    clb    Y0,X0
    asll.l X0,Y0    //normalize inum
    neg    X0    //set exponent
    rtsd
}

30
add.w #15,X0
move.w X0,A //exponent
move.w Y0,A0 //mantissa

//FFloat zero = 0xFF800000
Zero:
    rtsd
    move.w #$FF80,A
    clr.w A0
}

7.16 S32int2FFloat
//convert an int32 to an ffloat value
asm ffloat S32int2FFloat(long int inum)
{
    //inum = 0
    tst   A
    jeq   Zero

    //inum != 0
    clb   A,X0
    asll.l X0,A //normalize inum
    neg   X0 //set exponent
    add.w #31,X0
    rnd A
    rtsd
    move.w A1,A0 //mantissa
    move.w X0,A1 //exponent
    sxt.l A //sign-extend A to 36 bits

    //FFloat zero = 0xFF800000
Zero:
    rtsd
    move.w #$FF80,A
    clr.w A0
}

7.17 U32int2FFloat
//convert an unsigned int32 to an ffloat value
asm ffloat U32int2FFloat(long unsigned int unum)
{
    tst   A
    jeq   Zero //inum = 0
    jlt   LongUnsigned //If 2^31 <= unum <= 2^32-1, unum will
                     //be a negative number

    //unum <= 2^31 - 1
    clb   A,X0
    asll.l X0,A //normalize unum
    neg   X0 //set exponent
    add.w #31,X0
    rtsd
    move.w A1,A0 //mantissa
    move.w X0,A1 //exponent
    sxt.l A //sign-extend A to 36 bits
Zero:
  rtsd
  move.w #$FF80,A
  clr.w A0

// If unum is between 2^{31} and 2^{32}-1
LongUnsigned:
  lsr.w A // divide mantissa by 2
  move.w A1,A0 // move mantissa to its right place
  // divide the mantissa by two and increase the exponent by 1
  // this will correct the sign of A while keeping the absolute
  // value of a the same
  rtsd
  move.w #32,A1 // exponent will always be 32 for this case
  sxt.l A // sign-extend A to 36 bits

7.18 FFloatRnd2S16int
asm short int FFloatRnd2S16int(register ffloat ffnum)
{
  move.w A1,Y0
  move.w A0,A
  // Scale so that exponent = 15; converts mantissa to integer scale
  // Check if resulting mantissa is in range -32768 to 32767 (16 bit
  // signed int)
  sub.w #15,Y0 // Number is outside range -32768 to 32767
  jgt Over // Number is small and rounds to zero
  cmp.w #-17,Y0
  jlt Zero
  rtsd
  asll.l Y0,A // Scale to exponent = 15 (one word, two cycles)
  rnd A // Convergent rounding (round down boundary case
  // if even)
  move.w A1,Y0
  // end

Zero:
  rtsd
  clr.w Y0 // Result is zero
  nop
  nop
  // end

Over:
  tst A // branch to Neg: if number is below 32768
  blt Neg
  rtsd
  move.w #$7FFF,Y0 // Set to most positive 16-bit value
  nop // Filler for third delay slot
  // end

Neg:
7.19 **FFloatTrunc2S16int**

```assembly
asm short int FFloatTrunc2S16int(register ffloat ffnum) {
    move.w A1,Y0
    move.w A0,A

    //Scale so that exponent = 15; converts mantissa to integer scale
    //Check if resulting mantissa is in range -32768 to 32767 (16 bit signed int)
    sub.w #15,Y0
    jgt Over //Number is outside range -32768 to 32767
    cmp.w #-17,Y0
    jlt Zero //Number is small and rounds to zero
    rtsd
    asll.l Y0,A //Scale to exponent = 15 (one word, two cycles)
    move.w A1,Y0
    nop //Filler for third delay slot

    Zero:
    rtsd
    clrw Y0 //Result is zero
    nop
    nop //end

    Over:
    tst A
    blt Neg //branch to Neg: if number is below -32768
    rtsd
    move.w #$7FFF,Y0 //Set to most positive 16-bit value
    nop //Filler for third delay slot

    Neg:
    rtsd
    move.w #$8000,Y0 //Set to most negative 16-bit value
    nop //Filler for third delay slot

}
```

7.20 **IEEE2FFloat**

```assembly
asm ffloat IEEE2FFloat(register float fnum) {
    bftstl #$7F80,A1
    jcs Zero //For IEEE, zero is indicated by zero exp.
    move.w A1,Y0
    bfclr #$FF00,A1
    sxt.l A //Sign-extend A to 36 bits
    bfset #$0080,A1
```
`brclr`  #$8000,Y0,L1     //Branch if sign bit is positive
`neg`   A                  //Negate mantissa if sign bit is negative

L1:
  `clb`   A,X0              //Normalize mantissa
  `asll.l` X0,A
  `bfclr` #$807F,Y0
  `lsrr.w` #7,Y0
  `sub.w`  #119,Y0
  `sub`   X0,Y0            //FFloat exponent is ready
  `clb`   Y0,X0            //Check for overflow/underflow
  `cmp.w` #8,X0
  `jlt`   Exp_Err

Continue:
  `rnd`   A
  `rtsd`  
  `move.w` A,A0
  `move.w` Y0,A1
  `sxt.l` A               //Sign-extend A to 36 bits

//end

Zero:
  `RTSD`
  `MOVE.W` #$FF80,A
  `CLR.W` A0
  //END

Exp_Err:
  `cmp.w` #$007F,Y0        //If not overflow, go to underflow check
  `jle`   Underflow       //Positive or negative overflow?
  `tst.w` A1               //If negative, go to negative handler
  `jlt`   NegO
  `move.w` #$007F,A        //Max out exponent
  `rtsd`  //Delayed return from subroutine - will
  //execute next three words
  `move.w` #$7FFF,A0       //Max out mantissa
  `nop`    //filler for third delay slot
  //end

NegO:
  `move.w` #$007F,A        //Max out exponent
  `rtsd`  //Delayed return from subroutine - will
  //execute next three words
  `move.w` #$8000,A0       //Most negative mantissa
  `nop`    //filler for third delay slot
  //end

Underflow:
  `cmp.w` #$FF80,Y0        //Check for underflow
  `jge`   Continue         //Not an error
  `tst.w` A1               //Positive or negative underflow?
  `jlt`   NegU
  `move.w` #$FF80,A         //Minimum exponent
  `rtsd`  //Delayed return from subroutine - will
  //execute next three words
  `move.w` #$4000,A0        //Minimum normalized positive mantissa
  `nop`    //Filler for third delay slot
  //end

NegU:
  `move.w` #$FF80,A        //Minimum exponent
7.21 **FFloat2IEEE**

```c
float FFloat2IEEE(float ffnum) {
    float fout = 0;
    long int iexp = 0;
    long unsigned int tempout = 0, sign = 0, mantissa = 0, exp = 0;
    void *VoidPointer;
    float *FloatPointer;
    long unsigned int *LintPointer;
    if (ffnum&0xFFFF) //ffnum is not zero
    {
        mantissa = ffnum & 0x0000FFFF;
        exp = ffnum&0xFFFFF0000;
        iexp = (long int)exp;
        iexp += 0x0007F0000; //Bias exponent positive by 127
        if (iexp < 0x00010000) //Limit exponent size to allowed
            //IEEE range
        {
            iexp = 0x00010000;
        }
        else if (iexp > 0x00FE0000)
        {
            iexp = 0x00FE0000;
        }
        if (mantissa&0x00008000) //ffnum is negative
        {
            sign = 0x80000000;
            mantissa ^= 0x0000FFFF; //Negate
            mantissa++;        
        }
        while (!(mantissa&0x8000)) //normalize
        {
            mantissa <= 1;
            iexp -= 0x00010000;
        }
        if (iexp < 0x00010000) //Limit exponent size to allowed
            //IEEE range
        {
            iexp = 0x00010000;
        }
        else if (iexp > 0x00FE0000)
        {
            iexp = 0x00FE0000;
        }
    }
}
```
iexp = 0x00FE0000;

exp = (long unsigned int)iexp;
exp <<= 7; //Shift exponent to correct position
mantissa <<= 8; //Shift to correct IEEE position
mantissa &= 0x007FFFFF; //Clear leading one
tempout = sign | exp | mantissa;
else exp = 0x00000000; //zero

VoidPointer = &(tempout); //obtain pointer to unsigned long
FloatPointer = VoidPointer; //convert to float
fout = *FloatPointer;
return(fout);

7.22 FFneg
asm ffloat FFneg(register ffloat ffnum)
{
move.w A1,Y0 //store ffnum exp in Y0
move.w A0,A //A holds mantissa of ffnum
neg A //full 36-bit negate
asr A //shift right to prevent overflow of clb
jeq Zero //Don't normalize if zero

//ffnum != 0
clb A,X0 //Count sign bits
asll.l X0,A //Normalize
sub X0,Y0 //Adjust exponent
inc.w Y0 //Return to normal scale
clb Y0,X0 //check number of sign bits in exponent
cmp.w #8,X0 //If less than 8 (exp > 8 bits),
jlt Exp_Err //jump to exponent exception handler

Continue:
rtsd //delayed return from subroutine
move.w A1,A0 //Move mantissa of sum to lower word of ffnum1
//return value
move.w Y0,A1 //Move exponent to upper word of ffnum1 (return
//value)
sxt.l A //Sign-extend A to 36 bits
//end of main neg function

Zero:
rtsd //Delayed return from subroutine - will
execute next three words
move.w #$FF80,A //Set exp of sum to minimum
clr.w A0 //Set mantissa of sum to 0
//end of zero handler
Exp_Err:
  cmp.w #$007F,Y0 //If not overflow, go to underflow check
  jle Underflow //Positive or negative overflow?
  tst.w A1 //If negative, go to negative handler
  jlt NegO //Delayed return from subroutine - will
           //execute next three words
  move.w #$007F,A //Max out exponent
  rtsd //Delayed return from subroutine - will
        //execute next three words
  move.w #$7FFF,A0 //Max out mantissa
  nop //Delay slot filler
//end

NegO:
  move.w #$007F,A //Max out exponent
  rtsd //Delayed return from subroutine - will
        //execute next three cycles
  move.w #$8000,A0 //Most negative mantissa
  nop //Delay slot filler
//end

Underflow:
  cmp.w #$FF80,Y0 //Check for underflow
  jge Continue //Not an error
  tst.w A1 //Positive or negative underflow?
  jlt NegU //If negative, go to negative handler
  move.w #$FF80,A //Minimum exponent
  rtsd
  move.w #$4000,A0 //Minimum normalized positive mantissa
  nop //Filler for third delay slot
//end

NegU:
  move.w #$FF80,A //Minimum exponent
  rtsd //Delayed return from subroutine - will
        //execute next three words
  move.w #$BFFF,A0 //Minimum (abs) normalized negative
                   //mantissa
  nop //Filler for third delay slot
//end of E_Err

7.23 FFabs
asm ffloat FFabs(register ffloat ffnum)
{
  move.w A1,Y0 //store ffnum exp in Y0
  move.w A0,A //A holds mantissa of ffnum
  abs A //full-width absolute value
  asr A //shift right to prevent overflow of clb
  jeq Zero //Don't normalize if zero
  //ffnum != 0
  clb A,X0 //Count sign bits
  asll.l X0,A //Normalize
  sub X0,Y0 //Adjust exponent
  inc.w Y0 //Return to normal scale
clb Y0,X0  //check number of sign bits in exponent
cmp.w  #8,X0  //If less than 8 (exp > 8 bits),
jlt   Exp_Err  //jump to exponent exception handler

Continue:
  rtsd      //delayed return from subroutine
  move.w   A,A0  //Move mantissa of sum to lower word of ffnum1
                 //(return value)
  move.w   Y0,A1  //Move exponent to upper word of ffnum1 (return
                 //value)
  sxt.l A     //Sign-extend A to 36 bits
  //end of main abs function

Zero:
  rtsd      //Delayed return from subroutine - will execute
             //next three words
  move.w   #$FF80,A  //Set exp of sum to minimum
  clr.w    A0       //Set mantissa of sum to 0
  //end of zero handler

Exp_Err:
  cmp.w   #$007F,Y0
  jle    Underflow  //If not overflow, go to underflow check
  tst.w  A1        //Positive or negative overflow?
  jlt    NegO       //If negative, go to negative handler
  move.w   #$007F,A //Max out exponent
  rtsd    //Delayed return from subroutine - will execute
           //next three words
  move.w   #$7FFF,A0 //Max out mantissa
  nop     //Delay slot filler
  //end

NegO:
  move.w   #$007F,A //Max out exponent
  rtsd    //Delayed return from subroutine - will execute
           //next three cycles
  move.w   #$8000,A0 //Most negative mantissa
  nop     //Delay slot filler
  //end

Underflow:
  cmp.w   #$FF80,Y0 //Check for underflow
  jge    Continue  //Not an error
  tst.w  A1        //Positive or negative underflow?
  jlt    NegU       //If negative, go to negative handler
  move.w   #$FF80,A //Minimum exponent
  rtsd    //Delayed return from subroutine - will execute
           //next three words
  move.w   #$4000,A0 //Minimum normalized positive mantissa
  nop     //Filler for third delay slot
  //end

NegU:
  move.w   #$FF80,A //Minimum exponent
  rtsd    //Delayed return from subroutine - will execute
           //next three words
  move.w   #$BFFF,A0 //Minimum (abs) normalized negative mantissa
  nop     //filler for third delay slot
  //end of E_Err

}
8 References

[1] “56F8347 Data Sheet.” 14 December 2006
   <http://www.ortodoxism.ro/datasheets2/d/0jayk8l9f7lua3gy5gyglyjs4x3y.pdf>.


[3] “Two’s Complement.” Wikipedia, the free encyclopedia. 5 December 2006