Analysis of Manual ARM Floating-Point Arithmetic

Zachary Robinson
zach7@umbc.edu
Dean Fleming
deanf1@umbc.edu

Cory Ferrier
ferrier1@umbc.edu
Justin Sternberg
justins3@umbc.edu

Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County

8 December 2016

Abstract

This report describes a program written in ARM assembly that manually converts two inputted string-format signed floating-point numbers into single-precision IEEE 754 format, and performs addition, subtraction, and multiplication on them. The goal of writing this conversion manually was to compare the performance of the manual version with the built-in floating point operations. After testing, the manual versions described in this report reliably perform the operations faster than the built-in operations, but with some slight loss of precision.

Keywords: ARM, assembly, floating point, arithmetic, CPI, IEEE 754, ARMSIM

1. Introduction

IEEE-754 format is the foundation for floating point arithmetic in most modern processors. It serves an important role as a unified method by which different machines can expect to receive floating point values from each other. Despite this, its complicated and highly technical nature means that few have a proper understanding of how it functions and most are content to just take what their machine gives them.

Within the scope of designing manual arithmetic floating-point functions, it was our goal to gain insight into the strengths and limitations of writing code at the assembly level, specifically with respect to the level of control over the processor. With a focus on floating-point processing, we additionally hoped to gain insight into the limitations of representing the set of real numbers on limited physical memory.

The ARM, or Advanced RISC Machine, instruction set was used to develop our floating-point functions, containing a more limited, than a CISC architecture, set of generally applicable instructions. We were quickly able to adopt and understand the limited instruction set to author our own floating-point arithmetic. It was then possible to benchmark the success, with respect to precision of output and speed of execution, of our manual floating-point functions against the native ARM floating-point instructions in order to meet our goals of gaining further understanding the strengths and limitations of floating-point representation and writing functions in assembly.

2. Related Work

Critical to the portability of modern computer applications is the standardized form through which floating-points are manipulated, IEEE 754. Before the 1980's, many variations of floating-point representation were being used [5]. Because of this, like operations on different machines could generate significantly different results. Even among operations on the same machine, floating-points may have been interpreted differently [5]. Approaching 1980, Intel, with some encouragement, pushed to standardize floating-point representation across all of its processors [5]. In 1985, this goal was officially met, as the IEEE published its floating-point standards which, while updated in 2008, are still the standard today [6].

The single precision representation, which is referenced heavily within this report, includes 32 bits broken up into three critical segments, reflective of a number in binary exponential form [6]. The first segment, from left to right, consists of one bit that denotes the sign of the represented value, 0 being positive, 1 being negative. The second segment consists of the next 8 bits used together to represent an exponent in the range within the range [-126, 127]. By adding a bias of 127, the values directly represented by the 8 exponent bits are in the resulting range [1, 254]. While representable, exponents of -127, all bits unset, and of 128, all bits set, are reserved for special cases.

The last segment from left to right consists of the remaining 23 bits and represents the mantissa. The standard dictates that the mantissa always has only one digit to the left of the decimal place, and in any non-zero number, this digit will be 1. This is referred to as the "normal" form [4]. Because every non-zero floating-point has this leading 1, the IEEE 754 standard dictates that it is left out of the representation, allowing for an additional bit of numerical accuracy [6]. Because representing the number zero presents an exception to this, one of the reserved special cases mentioned before is used for zero representation.

3. Methodology

As discussed briefly in the introduction, we applied the ARM instruction set to develop our program. The program takes predefined ASCII input values and executes the following set of processes, including floating-point addition, subtraction, and multiplication. The program generates a result set of comparable values determined by the execution of each manual arithmetic function and its native ARM counterpart. It should be noted here that the manual functions developed process strictly single-precision floating-point values which are ultimately represented in standard IEEE 754 single-precision format, a 32 bit representation. The native ARM floating-point instructions were also applied to floating-points of the same single-precision standard such that the results could be readily compared.

The program was executed using an ARM simulator, specifically the ARMSim# application developed by members of the Department of Computer Science at the University of Victoria in Canada [1]. Without access to a dedicated ARM processor, the simulator allowed for consistent and reliable testing. ARMSim emulates the

ARM7TDMI-S, and has access to the same commands as the ARM7TDMI-S [2]. The instruction documentation for the ARM7TDMI-S was used as reference when implementing the program in ARMSim# [3].

The program can be broken down into the following steps:

1. Get input in the form of ASCII String
2. Convert ASCII Strings to Naive Format *
3. Convert Naive Format Results to IEEE 754-Split Format **
4. Perform Addition on IEEE 754-Split Format
5. Perform Subtraction on IEEE 754-Split Format
6. Perform Multiplication on IEEE 754-Split Format
7. Merge Results into traditional IEEE 754 Format
8. Calculate Results using Native Floating Point Instructions
9. Compare Results and Runtimes of Our Implementations vs Native Instructions

* Naive Format refers to storing the sign bit in one word, the whole number in the next word, and the fraction in the third word.

** IEEE 754-Split Format refers to a format where the sign bit is in one word, the 8-bit exponent is right aligned in the following word, and the 23 bit mantissa is right aligned in the last word.

Described below are explanations of each step:

3.1 Getting Input

Input is provided via the .asciz directive, which generates a null terminated string with the provided value. The inputs are given at the top of the .text section in the format of [+/-][whole number].[decimal].

3.2 Converting to Naive Format

This is handled via the getFirstBit, getWhole, and and getDecimal functions. getFirstBit takes the address of the word to store the sign word in, as well as the address of the string to check. It looks at the first character in the string and sets the sign word to 1 if the value is a ‘-’character, and zero otherwise.

getDecimal is then called and begins at the last character in the string. It reads the last character and subtracts 48 from it to get its integer value, then multiplies it by 1 and adds it to a running total. It then checks the second to last character, again subtracts 48 from it, and multiplies this value by 10 and adds it to the running total. It continues doing so until it reaches the decimal character, at which point it returns the location of the decimal character and stores the running total in the word meant to hold the decimal value.

getWhole then runs and follows the same process as getDecimal, the difference being that it starts 1 before the known location of the decimal point and terminates after it finishes processing the first character.

The nature of this design means that our program can successfully parse any given string representation of a value provided the whole number and decimal portions both fit into a 32-bit word.

3.3 Converting to IEEE 754 Format

The conversion of the parsed form of the string input is handled under the "getIEEE754" header. First, before branching to it, we load into registers the parsed number, the memory location for the resultant IEEE 754 number, and the memory location for the split version of the IEEE 754 number.

Before loading, the registers used in this algorithm are stored on the stack to keep the algorithm self-contained. Then, we load the parsed sign bit into the split IEEE 754. This is straight forward as the sign bit is the leftmost bit. Then, we load the whole number part of the parsed number into a register. If the whole number is a 1, we know our exponent is 0, so we can skip to the next step, "exponentInBin". If not, we calculate the exponent using the whole number. If the whole number is an n-bit number, then our exponent will be n - 1, so the exponent is calculated by counting how many bits our whole number is minus one. Then, we add 127 to the exponent, and then store that number in the split IEEE 754.

The next step is to calculate the mantissa. The whole number part of the mantissa is the most simple part. We take off the leading 1 bit of the whole number, using a number in the form 2^n that was created in counting the bit size of the whole number (for example, if our number is $14_{10} = 1110_2$, in R2 there is an 8 as a result of the counting algorithm, so subtracting 8 from 14 gives us $6_{10} = 0110_2$, which removes the leading one). After this, the whole number is stored in a register, and bit size of the whole number is subtracted from 23 to calculate how many bits we have left for the fraction part.

For the fraction part of the mantissa, we load in the fraction from the parsed number, and then count how many decimal digits the number is, using the same algorithm that counted bits, but this time with 10's instead of 2's. We do this because it was modelled after how you would calculate this on paper: You multiply the fraction by 2, and if you get a number in the form 0.xxx, you put a 0 and continue. If you get a number in the form of 1.xxx, you put a 1, subtract 1, and continue. If you ever equal exactly 1, you put a 1 and you are done. Since in our case our fraction is represented as a whole number (i.e. 4.695 parses the fraction as 695), we have to base this on numbers in the form of 10^x. For example, 695 * 2 = 1390, so we would put a 1, subtract 1000 and continue. this 10^x number ends up in R2 from the digit counting algorithm, and the code essentially works as you would expect on paper. In this case, however, we shift the whole number left once when we don't "overflow" into the next digit, and we shift left and add 1 when we do. This is looped for the number of spaces we have left in the mantissa. In the case we ever equal exactly the required 10^x number, we shift once, add the 1, and then shift left for however many spaces we have left. This essentially fills the remainder of the mantissa with 0's.

When the calculation is done, we store the mantissa in the last part of the split version, pop our registers from the stack, and return. Immediately after, we take the split version (which is only used for convenience) and push it all into one word. The algorithm that does this is explained in section 3.7.

3.4 Performing Addition

As indicated above, the addition function assumes that both values to be added have been broken down into IEEE 754-Split Format. With each floating-point segment, sign, exponent, and mantissa, separated, addition of the two values can be broken down into sequentially determining each segment for the sum. The first, and critical, sequence involves determining the sign, positive or negative, of the sum.

The function traverses one of two paths based upon comparison of the signs of the two input values. If both addends have the same sign, it is guaranteed that the result sign will be the same and a path, dubbed 'same_sign' is traversed in this case. A more complicated path, 'diff_sign' is traversed if it is determined that the signs of the addends are different. It can not yet be known what the resulting sign will be in this case, and so more complex processing is necessary. Before expanding on this process, let's first traverse the 'same_sign' path.

The 'same_sign' path is begun by storing the sign of the sum, since it has been determined. Next, the 'assumed' leading 1 of each mantissa, left out in IEEE 754 standard form, is added back in at the bit second from the left, creating a 24 bit mantissa. The path must then process both the exponent and mantissa of the two input values partially before determining their corresponding results in the sum. The second segment, the exponent, cannot be determined as simply as the sign and in some cases relies upon information gained from processing the mantissas. In the same way, the processing of the mantissas relies on information gained from comparing the exponents.

Since addition is commutative, the order of the two addends in input does not matter. Taking advantage of this fact, after the sign bit is stored, the relationship of the exponents of the two addends is determined with the goal of identifying the dominating (larger) exponent. If both exponents are equal, they and their associated mantissas are left alone and processing moves on to mantissa addition. If one exponent dominates, the dominating and subordinate exponents are placed into designated registers, along with their associated mantissas, regardless of input order. The difference in exponents is then calculated and used to shift the subordinate mantissa to the right within its 32 bit register. We note here that, with a difference in exponent greater than 7, precision could be lost as bits of the subordinate mantissa are shifted out of the word. This shifting is done to the subordinate mantissa to prepare for the adding of both addends, understanding that the amount of difference between exponents is a difference of powers of two, and each power of two is represented by one bit position.

Once the subordinate mantissa has been adjusted, both mantissas are prepared for adding. The basic ARM add instruction is used to add the two mantissas. The leftmost bit of the resulting word is checked and if flagged, 1 is added to the exponent of the sum, reflective of the increase in bit position by 1. The mantissa is then put back into its IEEE 754-Split Format and stored along with the exponent, completing the determination of each segment of the sum.

The more complicated path, ‘diff_sign’ begins by determining the sign segment of the sum. It does so by taking the sign bit of the larger addend, first comparing the exponents, and then if necessary the mantissas, of each addend. In the process, the dominating (larger magnitude) addend is identified if there is one, along with its sign. Similar processing to that in ‘same_sign’ is then executed, adding back in the ‘assumed’ 1 to each mantissa and shifting the subordinate accordingly if necessary. If at this stage it is determined that the two addends are of equal magnitude, the result is known to be 0 and processing is complete. The subtraction instruction is then applied to the two addend mantissas, retaining order according to which addend was negative, with the understanding that addition between a negative and positive value will result in a sum with a lesser absolute value.

The final bit of the ‘diff_sign’ path examines the bits of the difference between the mantissas in order to identify any reduction in exponent. Where an add of two same-sign addends can result in no more than an increase of 1 to the dominating exponent, an add of two different-sign addends can result in a much larger decrease to the dominating exponent, potentially down to 0. The leading bit of the difference of mantissas is identified and its distance in bit positions from that of the dominating mantissa is subtracted from the dominating exponent. Afterwards, the exponent and mantissa of the sum are put back into IEEE 754-Split format and stored.

3.5 Performing Subtraction

The subtraction function follows the same potential paths with the same processing as the addition function, ‘same_sign’ or ‘diff_sign,’ but each with the opposite functioning of its counterpart in addition. The ‘same_sign’ path in subtraction executes as the ‘diff_sign’ path in addition and conversely, the ‘diff_sign’ path in subtraction executes as the ‘same_sign’ path in addition. This is the case because subtraction between two different-signed values results in an increase in magnitude of result, while subtraction between like-signed values results in a decrease in magnitude of result, the opposite of addition.

3.6 Performing Multiplication

We first determine the sign bit of our result by doing an XOR command on the sign bit of both values. We store this result.

We then determine the exponent by adding the exponents of the 2 values together and subtracting a bias of 127. We do not store this result in memory yet because it might change after we calculate the mantissa.

We begin calculating the mantissa by loading in the mantissa of both values. We then add in an assumed one onto both mantissas.

A running total for the mantissa is initialized to zero. The least significant bit is checked in the second value, if it is a 1 we add the first value to our running total. We shift our running total right by 1 and do the same the to the second value. We continue to do this until the second value is zero (all bits have been shifted out) at which point we exit our loop.

Afterwards, we check to see if our mantissa resolved as a 24 bit value, in which case we must add 1 to our exponent and shift our mantissa right by one. Finally we shift out our assumed 1 and align our mantissa into the right 23 bits. We now store the exponent and mantissa and return.

3.7 Merging Results

The IEEE754 split values are provided for the result of our multiply, add, and subtract functions. The 1 bit left-aligned sign is added to the 8 bit exponent, which is shifted right by 1 bit to make room for the exponent. The right-aligned 23 bit mantissa is then added in and the resulting value is returned.

3.8 Calculating Results using Native Floating Point

The IEEE754 representations of both inputs are loaded into the floating point registers. The fadds, fsubs, and fmuls operations are then performed on these values and the results are returned to the general purpose registers. These values are then written to memory for comparison.

3.9 Comparing Our Results to Native

We compare the results of our function to the native implementation in regards to both running time, instructions count and accuracy.

To measure the running time, we set ARMSim# breakpoints before and after we make an add/subtract/multiply function all. ARMSim# displays the execution time and number of instructions performed between the two breakpoints in the output view. It is worth noting that the floating point instructions will always return an instruction count of ‘1’, so we only record the execution time for those calls.

We compare the accuracy of our results to the native implementation by observing our results in the memory view and noting any difference. This is made simpler because we store our results in the following format:

32 Bits	32 Bits	32 Bits	32 Bits	32 Bits	32 Bits
Manual Add Result	Native Add Result	Manual Subtract Result	Native Subtract Result	Manual Multiply Result	Native Multiply Result

4. Discussion

The following section details our experimental results, including an analysis of CPI, or cycles per instruction, running time, and error.

4.1 CPI Comparison

Below are the tables used in our calculations of CPI for an average input for each function, including the instruction count and the cycles for each instruction [8].

Multiplication:

Instruction	Count	Cycles
ldr	6	3
str	3	2
add/sub	26	1
mov	80	1
branch/cmp	23	3

Total Instructions = 138
Total Cycles = (6*3 +3 * 2 + 26 + 80 + 23*3) = 196
Manual CPI = Total Cycles / Total Instructions = 1.44
Native CPI: 3/1 = 3.0

Addition and Subtraction:

Instruction	Count	Cycles
ldr	18	3
str	14	2
add/sub	23	1
mov	51	1
branch/cmp	39	3

Total Instructions = 145
Total Cycles = (18*3 +14 * 2 + 23 + 51 + 39*3) = 273
Manual CPI = Total Cycles / Total Instructions = 1.88
Native CPI: 2/1 = 2

Putting these results into a table, we get:

	Manual Multiply	Native Multiply	Manual Add/Sub	Native Add/Sub
CPI	1.44	3	1.88	2

Based on this data, we can conclude that our functions have a lower overall CPI than the native versions of the same instructions. With this we can anticipate that our functions, barring any abnormalities, will on average perform better assuming that the number of instructions is comparable

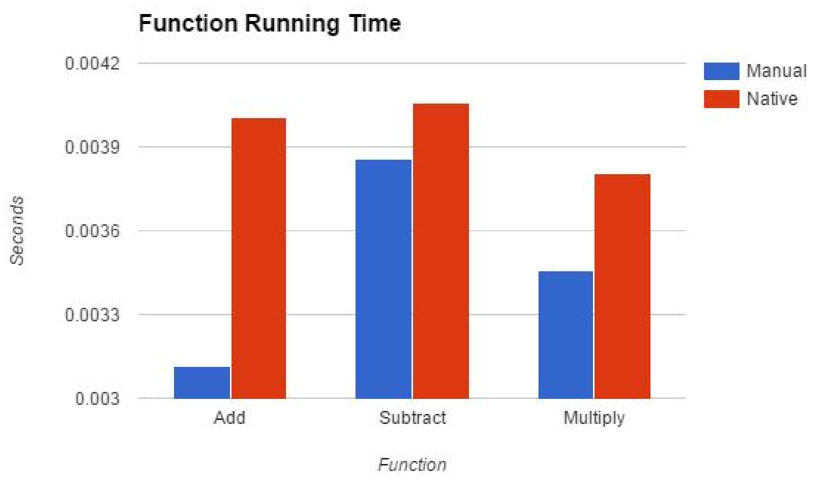
4.2 Sample Results

Included below are the results of a few sample runs. Note that this is different than the data used in section 4.3, where the difference between the native and actual results were taken as well as the running times.

Input 1	Input 2	Manual Add	Native Add	Manual Subtract	Native Subtract	Manual Multiply	Native Multiply
-1.2	+1.8	CF333334	CF333334	C0466666	C0466666	C011EB80	C011EB84
-1023.123	-235.7	C49D5A55	C49D5A55	C444DB12	C44DB12	486B7F80	486B7F85
-4.0	-4.0	C1000000	C1000000	00000000	00000000	41800000	41800000
-7.81875	+65.0	4264B999	4264B99A	C291A333	C291A333	C3FE1BFC	C3FE1C00

4.3 Performance Analysis

When benchmarked against the native ARM floating-point arithmetic instructions, our manual functions actually executed faster on average. Over the range of inputs tested, the manual addition function measured the greatest average execution time improvement, nearly a full millisecond, over its native counterpart. The following graph depicts the results for running times.



Both the manual subtraction function and multiplication function also outperformed their native counterparts by about 0.2 and 0.35 milliseconds respectively. These results were surprising at first, but upon further consideration seem justifiable. Our manual arithmetic functions sacrifice a small level of precision for their speed increase, as the native floating-point instructions perform a more detailed rounding process. Within the manual functions, bits of least significance in the mantissa of each floating-point value are lost entirely when they no longer fit into the allotted 23 bits. Additionally, there is likely more overhead cost for invoking the native floating-point instructions than the manual functions, as the manual functions operate within the standard registers.

Losing bits of any significance will result in a loss of precision. As the native ARM instructions are a published and industry quality set of instructions, the results of the native floating-point operations were used to assess error in the results of the manual functions. One way to measure this is to take the magnitude as the decimal difference between the native floating-point instruction result and the manual result. Using this method, the average error measured across each set of inputs and function type was about 0.064. While for some applications this level of error might be tolerable, more critical applications would require significantly less error. Consider an application involved in processing money, for example. It would not that take many iterations of an error of $0.06 to result in significant loss.

When broken down by type of function, addition, subtraction, or multiplication, it can be seen that most of the error can be attributed to only one of the three. Only 1 of 10 trials of addition resulted in any error, resulting in an average of only 0.000004, significantly less than the 0.064 average for multiplication. Subtraction generated slightly more error than addition, measured in 4 of 10 trials for an average of 0.000382, still significantly less than the average of multiplication. With this knowledge, it is clear that the largest contributor to overall error was the multiplication function. Error was measured in 8 of 10 input trials for multiplication, resulting in an average magnitude of about 0.1915.

Assessing error by examining bits, instead of decimal magnitude, indicates more of a closeness in solution, especially for multiplication, of the manual to the native function. In nearly all cases of error, 13 of the 30 calculations, the difference in the two results was only of one bit of mantissa. As suggested before, it is likely that the increase in execution time of the native functions when compared to the manual is justifiable so that results are guaranteed to be more accurate.

5. Conclusion

Writing and testing these floating-point functions increased our knowledge and understanding of the assembly writing process, as well as the benefits and limitations of floating point representation. As our manual functions experienced an improvement in execution time upon the native, they also suffered from a decrease in precision. This is reflective of the nature of floating-point representation. As one increases the precision of the representation of a true value as a floating point, one sacrifices speed of processing. While the scope of this experiment dealt only with single precision floating-point values, this tradeoff is easily understood by examining double precision floats. The number of bits for representation is doubled, greatly increasing precision, but also significantly increasing processing time, as twice as many bits must be manipulated. Future work will focus on improving the level of accuracy for single precision representation by implementing intelligent rounding so that results will be more useful in practice.

References

[1] ARMSim. (n.d.). Retrieved December 08, 2016, from http://armsim.cs.uvic.ca/

[2] ARMSim# version 2.1 for Windows. (n.d.). Retrieved December 08, 2016, from http://webhome.cs.uvic.ca/~nigelh/ARMSim-V2.1/index.html

[3] ARM Information Center: ARM7TDMI-S Technical Reference Manual. (n.d.). Retrieved December 08, 2016, from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0234b/BGEJCAFI.html

[4] Adding Floating Point Numbers. (n.d.). Retrieved December 08, 2016, from http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BinMath/addFloat.html

[5] An Interview with the Old Man of Floating-Point. (n.d.). Retrieved December 08, 2016, from https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story

[6] IEEE Standard for Floating-Point Arithmetic. (2008, August 29). Retrieved December 8, 2016, from http://www.csee.umbc.edu/~tsimo1/CMSC455/IEEE-754-2008.pdf

[7] Schmidt, H. (n.d.). IEEE 754 Converter. Retrieved December 08, 2016, from https://www.h-schmidt.net/FloatConverter/IEEE754.html

[8] ARM Information Center: Cortex™-A8 Technical Reference Manual. (n.d.). Retrieved December 08, 2016, from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344b/BABBGJHI.html