Converting floating-point applications to fixed-point

By Randy Allen
EEdesign.com
(09/24/04, 06:23:00 PM EDT)

Digital signal processors (DSPs) represent one of the fastest growing segments of the embedded world. Yet despite their ubiquity, DSPs present difficult challenges for programmers. In particular, because computation speed is critical to DSP applications, DSPs as a rule focus on supporting fixed-point operations.

This focus means that programmers must not only deal with mathematically sophisticated applications, but they must also deal with the errors introduced by performing these applications using reduced-precision arithmetic. This type of error analysis is yet another subject to be mastered by DSP programmers.

While it would be ideal if programmers could avoid using fixed-point arithmetic, as a practical consideration, they cannot. Fixed-point DSPs execute at gigahertz rates; floating-point DSPs peak out in the 300-400 megahertz range.

Because fixed-point DSPs are consumed in large volumes, their price per chip is a fraction of the price of a floating-point DSP. As a result, the only developers who can reasonably justify using a floating-point DSP are those developing low-volume applications requiring high precision arithmetic.

While converting floating-point applications to fixed-point appears daunting, the task often suffers from "fear of the unknown" syndrome. With knowledge of the issues, the right tools, and a well-thought out development methodology, the conversion process is very manageable.

Without knowledge of the issues, the right tools, and a well-thought out development methodology, floating-point applications can in fact suffer the same problems as fixed-point equivalents. The reduced precision and range of fixed-point numbers means that fixed-point applications encounter stability problems more easily than floating-point applications, but both types of applications encounter them.

$$\begin{pmatrix} 1.0 & 0.9 & 0.8 \\ 1.1 & 1.0 & 0.9 \\ 1.2 & 1.1 & 1.0 \end{pmatrix} \times \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}$$

Figure 1 — Ill-conditioned problem

To illustrate more concretely, consider the set of simultaneous linear equations in Figure 1. High school algebra teaches one way of solving simultaneous equations: add and subtract multiples of rows and columns to eliminate variables from positions until you get an easily-solved system (an upper triangular matrix).

Solving the system this way yields a result of x = [-14 138 134]. Alternatively, you can compute the left-inverse of the matrix and multiply that by the vector on the right. The analog to this solution in the scalar world (that is, where the left operand is a scalar rather than a matrix) is computing 1/A (where A is the left operand) and multiplying both sides of the equation by that quotient.

The left side is reduced to just x; the right side will be the solution. That method yields a solution of x = [-41.75 27.0 -78.5] — very different from the first. This difference is not just a function of these two methods; other methods will yield different answers as well. The difficulty is the problem itself.

Figure 1 illustrates an "ill-conditioned problem." Ill-conditioned problems are problems that are "close" or "nearby" other problems whose solutions are not nearby the true mathematical solution of the original problem. So, a slight change in the problem (as caused, for instance, by round-off error while solving) can produce an answer that's nowhere close to the solution of the original problem.

Ill-conditioned problems require care to solve even when in double-precision arithmetic. The message of ill-conditioned problems is that properties of a numerical algorithm require close study, regardless of whether the algorithm is implemented in floating-point or in fixed-point. Fixed-point arithmetic by definition has less precision and thereby more error than floating-point arithmetic. Less precision triggers instability more easily, but instability can arise using floating-point or fixed-point.

Understanding how fixed-point arithmetic triggers stability issues requires a quick review of fixed-point arithmetic. "Fixed-point" arithmetic is a phrase that encompasses three different forms of arithmetic formats: integer arithmetic, fractional arithmetic, and mixed arithmetic. Integer arithmetic is the fundamental building block for all other formats (including floating-point arithmetic). In integer arithmetic, bits are interpreted as the 2's-complement representation of an integer. Figure 2 illustrates a 4-bit integer number.

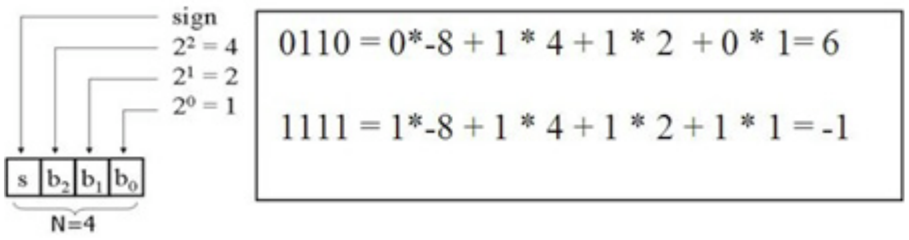

Figure 2 — Interpretation of bits in a 4-bit integer

Integers may be either signed or unsigned. In the case of an unsigned integer, the leading bit is not treated as a sign bit, but instead a multiple of the next higher power of 2 (in the example of a 4 bit number in Figure 2, the lead bit multiplies +8 rather than -8).

The range of numbers which an integer can represent is dependent on the number of bits N in the integer. For unsigned numbers, the smallest possible value is 0; the largest possible value is 2^N-1. For signed numbers, the smallest possible value is -2^{N-1}; the largest is $2^{N-1}-1$. The smallest resolution for an integer is one.

Integers are perfect for representing integral values, but most real world quantities require finer resolution. That requirement drives the second format: fractional arithmetic. Fractional arithmetic uses the same integer quantities, but interprets them as a fraction by assuming that the decimal point (technically, a "binal point") is inside the quantity rather than at the right.

In the case of unsigned fractional numbers, the binal point is implicitly at the left (since there is no need for a sign bit). Signed fractional numbers must reserve one bit to represent the sign, placing the binal point one bit from the left. Figure 3 provides more details on the fractional representation.

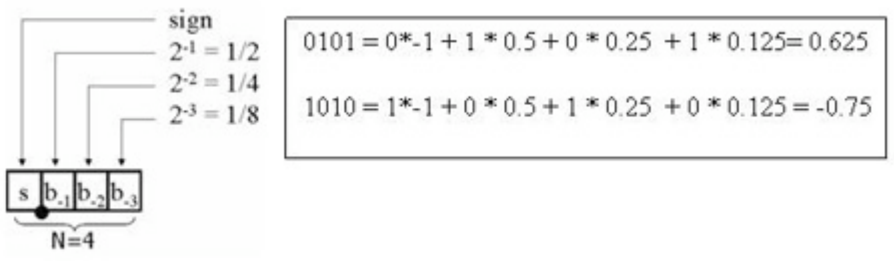

Figure 3 — Interpretation of fractional numbers

Fractional numbers obviously provide a more nearly continuous set of values than integers, and thereby provide more resolution. The resolution is not free; fractional numbers have a more limited dynamic range, since fractions are limited to having an absolute value that is less than or equal to 1.

The obvious compromise is to combine the two into a mixed format. In such a format, the binal point is not set at any specific location within the number; it is instead allowed to be anywhere within (or, for that matter, outside) the number proper. This flexibility allows the programmer to trade resolution versus range by choosing the binal point placement. Again, however, nothing is free. Generally, the programmer must manually track the binal point location. While not difficult (particularly to those who grew up in the era of slide rules), it is tedious.

Mixed format provides a balance between integers and fractionals; a programmer can trade bits to obtain either more range or more resolution. The dynamic range depends on the number of integer bits as well as the total number of bits in the quantity. For I integer bits, a signed mixed mode number will range between -2^{I-1} and $2^{I-1}-2^{-(N-I-1)}$; an unsigned number will range between 0 and $2^{I}-2^{-(N-I)}$.

Mixed mode incurs another cost. With the exception of fractional multiplication, which requires a little bit of extra maneuvering, both integer and fractional arithmetic can be affected using standard integer operations.

Mixed mode is not so straightforward. Adding a mixed mode number with 5 integer bits and 3 fractional bits to a mixed mode number with 4 integer bits and 4 fractional bits requires shifting to align the binal point prior to doing the addition proper. Forgetting to align (easy to do given that you have to manually track the binal point) creates frustrating errors.

Fixed-point processors gain speed and power efficiency over floating-point processors at the cost of reduced precision, reduced range, and increased headaches resulting from manual binal point tracking. Fixed-point requires that programmers find a fixed-point representation small enough to execute efficiently (the fewer the bits, the more speed and power efficient the application) but that maintains enough range and precision to execute successfully.

Note that "execute successfully" means more than just avoiding instabilities in the algorithm. Most signal processing involves image files or sound files that are evaluated by human senses. A precision loss that is not detectable by human senses is certainly acceptable; greater losses in precision may not be so.

Additionally, signal processing algorithms are mathematically complex, and need exploration and understanding at the floating-point level. Once the floating-point properties are understood and a stable algorithm has been created, an understanding of the data is required.

Balancing range and resolution in the fixed-point format requires a thorough knowledge of the range of the data. Once those characteristics are understood, a prototype implementation is next. Issues such as getting the proper shifts for addition of fixed-point numbers and tracking the binal point of mixed-mode numbers are not difficult, but they are tedious.

Tracking these issues through a detailed implementation is difficult; it is much better to start at a higher level that has less detail. In other words, time spent speeding up an incorrect algorithm is not time productively spent.

These requirements have led to a fairly uniform methodology for DSP applications, illustrated in Figure 4.

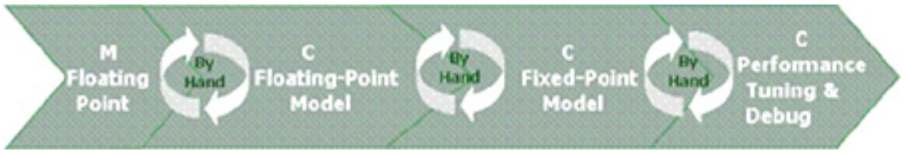

Figure 4 — DSP application development flow

While not universal, this flow is the choice of a large number of DSP development teams. Initial prototyping is done in floating point in M, the language of the Matlab interpreter1. M is an excellent language for high level prototyping and exploration.

After initial exploration and prototyping, the algorithm is manually converted into a floating-point C application. The conversion is necessary because the next step involves understanding the data, which in turn requires a large amount of simulation.

The Matlab interpreter, like any interpreter, is slow compared to compilers. It cannot provide the simulation speed necessary to simulate large computations. Since C is compiled, it provides the speed required to do enough simulation to understand the fixed-point precision requirements.

Once these characteristics are understood, a third manual conversion — this time to fixed-point C — is undertaken. C provides some support in terms of tracking binal points and getting fixed-point results, and all DSPs provide support for C compilers. C is not an ideal language in that it does not directly support some features such as saturation arithmetic (all C arithmetic is modulo by default) that are required on DSPs, but it is a much easier development language than assembly. This fixed-point model then serves as golden reference for the final implementation.

This development methodology, while common, is far from ideal. The same application is rewritten by hand four times. Because the initial development in M is in floating point, getting reference results for the final implementation is difficult. Changes in specification required from fixed-point problems are not uncovered until the third model is developed; feeding those changes back into the beginning of the development cycle is time-consuming, painful, and error-prone.

Because there are 4 different models written in 4 different languages, communication is difficult. Basic C provides little support in terms of tracking binal points; C++ can help this, but at a price — the resulting implementation code is less efficient. Even with these flaws, this methodology is a proven way of developing DSP applications.

Technology advances can eliminate many of these flaws. At least two companies now provide fixed-point support for M. This support allows the exploration and graphical facilities of M to be applied to fixed-point programs in addition to floating-point programs, making it easier to diagnose reduced-precision problems.

M also enables automatic tracking of binal points, eliminating a major headache. The result of such exploration can be a golden M model that produces bitwise identical results to the application running on a DSP.

Unfortunately, fixed-point support in M is not enough to solve all the fixed-point exploration problems. As mentioned previously, simulation is important for verifying that a quantization meets the "human perception" test. Interpreted execution cannot provide enough simulation cycles to perform that test.

Compiled simulation is a fundamental technology requirement. Similarly, you cannot explore fixed-point characteristics without knowing the types of various operands and operations (that is, are they fixed-point or floating-point), again a technology not currently supported in M.

With these technical advances, however, the design flow illustrated in Figure 5 becomes feasible. In such a flow, all exploration — floating-point and fixed-point — is performed in M, resulting in one golden M model that provides results that are bitwise identical to those arising from the implementation. Compilation technology can then automatically convert that fixed-point model into an implementation-level model tuned to a particular processor.

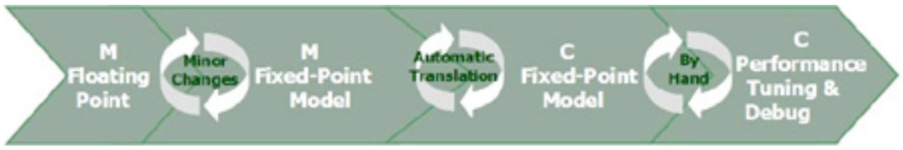

Figure 5 — Improved design flow

Using a fixed-point DSP for a signal processing application is more difficult than using a floating-point DSP, but not that much more difficult. Running an application in fixed-point is not such a fearful event, particularly if the proper analysis has been performed to verify the floating-point algorithm up front. When this analysis is extended to fixed-point, fears of overflows and underflows are unfounded.

Given the cost and performance advantages of fixed-point DSPs, there are strong rational reasons to balance against the unfounded numerical fears for converting to fixed-point processors. And, as chips decrease in size and correspondingly increase in speed, the "application" portion of Application Specific Integrated Circuits (ASICs) is becoming far more sophisticated mathematically. Increased mathematical sophistication means increased exploration and verification at both floating-point and fixed-point representations. This will expand the need for improved methodologies in ASIC and FPGA development.

Randy Allen, Ph.D., is founder and CEO of startup Catalytic Inc., which provides a tool that helps with floating-point to fixed-point conversion. Allen has more than 20 years of software and compiler development experience in startup companies, advanced technology divisions, and research organizations. His previous management experience includes serving as vice president of engineering at CynApps, vice president of performance engineering at Chronologic Simulation, executive director of software development at Kubota/Ardent Computer, and director in the Advanced Technology Group at Synopsys.