Application Note

Fixed-Point Arithmetic in Impulse C

Ralph Bodenner, Senior Applications Engineer
Impulse Accelerated Technologies, Inc.

Overview

This document describes the current level of support for fixed-point arithmetic in Impulse C and provides examples of how to implement fixed-point arithmetic using Impulse C datatypes and operations.

What is Fixed-Point?

Fixed-point arithmetic is a method of representing and calculating real numbers using integer datatypes and hardware. Fixed-point is an alternative to floating-point, a more well-known method of representing real numbers. Floating-point offers a greater range of values and more precision, but is also significantly more expensive (in terms of computation time and hardware required) than integer or fixed-point math. Embedded systems and digital signal processing (DSP) designers often choose fixed-point in order to achieve greater speed and reduce hardware cost in their designs.

There is, as yet, no commonly-accepted standard for representing fixed-point numbers. The SystemC and Embedded C standards both define fixed-point types and operations, but it is still up to the programmer to determine the parameters of computations, including precision, rounding and saturation, on an application-by-application basis. Floating-point, on the other hand, is codified in IEEE Standard 754, which defines the binary representation of single- and double-precision floating-point datatypes, as well as behavior of arithmetic operations across the entire range of representable values.

Fixed-point representations generally divide a fixed-width bit field into three parts: sign *(S)*, integer *(I)*, and fraction *(F)*. This document uses the following notation to denote a fixed-point format: *S*s*I.F.* For example, a fixed-point number with 1 sign bit, 8 integer bits, and 23 fractional bits is a 1s8.23 number:

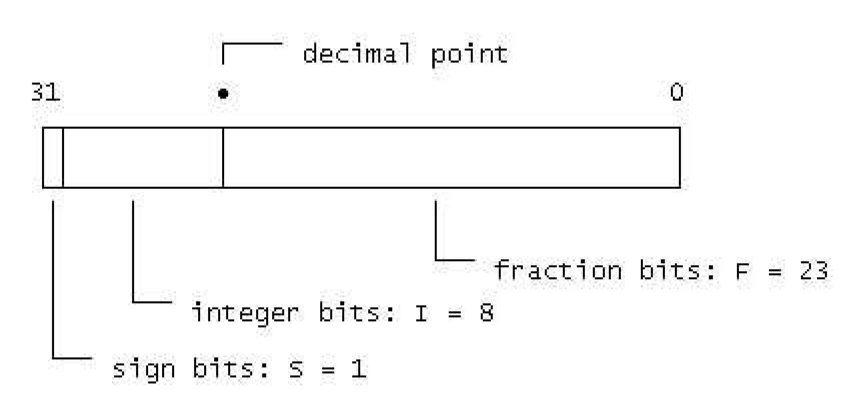

Figure 1: *Layout of a 1s8.23 fixed-point number*

The value of a signed fixed-point number is usually given by interpreting the bit field as a two's-complement binary number with a decimal point between *I* and *F*. The precision of a fixed-point number, or the granularity to which it is accurate, is equal to 2^{-F}.

The sizes of the bit field and of the integer and fractional parts are application-dependent, which helps explain the lack of a standard fixed-point representation. Designers must juggle these three parameters and come up with an appropriate combination, one that that meets each variable's requirements for datapath size, numerical range, and precision. The choice of whether a given fixed-point number is signed or unsigned is also left to the designer. For example, a 16-bit variable representing a normalized sine wave will only ever assume values in the interval (-1, 1). To cover this entire interval with greatest precision, a designer would allocate all but two bits to the fractional part, leaving one sign bit and one integer bit. This variable's format is 1s1.14, which can represent values in the interval [-3, 2] with precision 1/16384.

Creating Fixed-Point Applications

Fixed-point applications are often created from a well-tested floating-point implementation, rather than written from scratch. The process of converting a floating-point application to fixed-point is a non-trivial effort, with many issues to consider. In most cases, the floating-point program is instrumented with code that tracks the precision and range of the variables as it is run with sample data that simulate real-world inputs. The precision and range data obtained in this simulation step are used to determine the variables' fixed-point formats.

As a designer, you must convert each variable initially to a fixed-point format and keep track of that format as the variable is operated on. Each arithmetic operation has the potential to disrupt the tidy packaging of your variables. To begin with, adding or subtracting fixed-point numbers *A* and *B* requires that the decimal points be aligned—that is, $F_A = F_B$. Aligning the decimal points is usually accomplished by right-shifting one of the operands, which discards precision.

More troublesome is the possibility of overflow. Add a pair of 1s2.5 numbers and it's possible that the result will require three integer bits (e.g., 01100000 + 01000000 = 10100000), an incorrect result when interpreted in the 1s2.5 format. What to do? You can choose either to prevent overflow or to use saturation to put a ceiling (or floor) on a calculation. In the case of addition, overflow is prevented by shifting the decimal point of the operands to the right one bit before adding them, but you'll lose one digit of precision in the process.

Preventing overflow with the other basic arithmetic operations also affects the precision and range of their results. Multiplication can easily overflow, since the result can require almost twice as many bits as the operands have. The result must be right-shifted back into the desired format; if the multiplier hardware does not support double-precision intermediate results, you may lose integer bits. Again, scaling down the operands can prevent overflow but costs precision. Fixed-point division chops off least-significant bits in the result, but pre-scaling the operands (left-shifting this time) can preserve precision and prevent underflow, at the cost of reduced integer range in the operands.

Some CPUs and many DSPs provide hardware support for fixed-point arithmetic. To help prevent loss of integer range in multiplication, for example, a CPU might offer a double-precision intermediate result. If your hardware supports it, you can use saturation, which instructs operations to return the maximum (or minimum) value for a datatype when overflow occurs. To mitigate the loss of precision associated with negatively scaling variables (shifting them right), you can use rounding modes supported by your hardware to carry information from the lost bits back into the scaled number.

It's important to remember that you, the programmer, must keep track of where the decimal point is throughout a fixed-point calculation in order to make sense of the results. It's unlikely that a single fixed-point format will be used for every variable in a calculation, so fixed-point programs will be full of scaling operations to align decimal points, prevent overflow, and manage precision. The decision to scale variables depends on the actual values the variables are expected to take on, so it's important to be able to characterize the range and precision of input, intermediate, and output variables throughout a fixed-point program. Converting floating-point applications to fixed-point is an inherently time-consuming process that is beyond the scope of this document to describe in full, but several techniques are detailed in papers and manuals easily found by an Internet search (see Further Reading, below).

Impulse C Fixed-Point Macros

Impulse C provides support for fixed-point arithmetic in the form of macros and datatypes that allow you to express fixed-point operations in ANSI C and perform computations either as software on an embedded CPU or as hardware modules running in an FPGA's logic.

Impulse C currently supports three fixed-point bit widths (8, 16, and 32 bits) through a combination of datatypes and arithmetic macros. The `co_int8`, `co_int16`, and `co_int32` datatypes are provided for signed fixed-point numbers, while `co_uint8`, `co_uint16`, and `co_uint32` are for unsigned fixed-point numbers. When used with the appropriate class of macros, operands of a given type will be translated by the CoBuilder hardware compiler into a datapath of the same bit width, with two exceptions: for division and multiplication operations, CoBuilder will generate intermediate datapaths twice the size of their operands.

The Impulse C fixed-point macros are defined in the C header file `co_math.h`. Each macro takes two or three arguments: one or two operands *of the same fixed-point format* (`a` and `b`) and one constant integer (`DW`) whose value is the fractional bit width F_{abc} of the operands and the result. The programmer is responsible for pre-scaling the operands appropriately to prevent overflow or underflow. Macros for formatting fixed-point numbers, converting fixed-point values to floating-point, and performing fixed-point arithmetic are described in the following sections.

Formatting

To convert an integer value to a fixed-point format, use the `FXCONST8`, `FXCONST16`, or `FXCONST32` macros. The result is an unsigned integer (`co_uint8`, `co_uint16`, or `co_uint32`, respectively) with the given fractional bit width; you may cast the result to obtain the desired bit width and sign.

For example:

```
co_int16 a = (co_int16) FXCONST16(96, 7);
// 96 in 1s8.7 format == 0x3000
```

Converting Fixed-point to Floating-point

To convert a fixed-point number to floating-point, use the `FX2REAL32` macro. The result is a single-precision floating-point number (`float`) equal in value to the fixed-point number.

For example:

```
IF_SIM(
co_int32 a = 0x000000F4;  // 15.25 in 1s11.4
float f = FX2REAL32(a, 4);
printf("int: 0x%x, float: %f\n");
// prints "int: 0xF4, float: 15.250000"
)
```

The `FX2REAL32` macro is useful for debugging fixed-point values as they pass through a computation. Note that since floating-point numbers are not supported in hardware processes, this macro will only compile in simulation and in software processes targeting processors with floating-point hardware.

Addition/Subtraction

The Impulse C macros `FXADD8`, `FXADD16`, and `FXADD32` implement fixed-point addition.

For example:

```
co_int16 a, b, c;
a = 0xFF00;            // -1.0 in 1s7.8
b = 0x0180;            //  1.5 in 1s7.8
c = FXADD16(a, b, 8);  // 0x0080 == 0.5 in 1s7.8

co_uint8 x, y, z;
x = 0xFD;              // 63.25 in 0s6.2
y = 0x02;              //  0.5  in 0s6.2
z = FXADD8(x, y, 2);  // 0xFF == 63.75 in 0s6.2
```

Multiplication

The Impulse C macros `FXMUL8`, `FXMUL16`, and `FXMUL32` implement fixed-point multiplication, rounding the result to the nearest representable number. The macros use a double-precision intermediate datapath and return the low-order half of the result; if 64-bit integers are not supported on your target software platform, then multiplication of 32-bit fixed-width numbers will also not be supported in software.

For example:

```
co_int32 a, b, c;
a = 0x00002000;  // 32.0 in 1s23.8
b = 0x80000080;  // -0.5 in 1s23.8
c = FXMUL32(a, b, 8);  // 0x80001000 == -16.0 in 1s23.8
```

Division

The Impulse C macros `FXDIV8`, `FXDIV16`, and `FXDIV32` implement fixed-point division, rounding the result to the nearest representable number. The macros use a double-precision intermediate datapath and return the low-order half of the result; if 64-bit integers are not supported on your target software platform, then division of 32-bit fixed-width numbers will also not be supported in software.

For example:

```
co_int16 a, b, c;
a = 0x1000;             // 4.0  in 1s5.10
b = 0x0100;             // 0.25 in 1s5.10
c = FXDIV16(a, b, 10);  // 0x4000 == 16.0 in 1s5.10
```

Further Reading

For more information on fixed-point arithmetic and floating-to-fixed-point conversion:

- "Fixed-point math in C" (www.embedded.com/showArticle.jhtml?articleID=15201575)
- "An Introduction to Fixed Point Math" (www.bookofhook.com/Article/GameDevelopment/AnIntroductiontoFixedPoin.html)
- "Performance improvement using Fixed-Point arithmetic" (www.einfochips.com/download/tip_march04.htm)
- "Fixed-Point Literature" (www.ert.rwth-aachen.de/Projekte/Tools/FRIDGE/FixedPointLiterature.html)