Bringing floating-point math to the masses

By Geir Kjosavik
Sr. Staff Product Marketing Engineer
Embedded Processing Division
Xilinx Inc.
E-mail: geir.kjosavik@xilinx.com

Inside microprocessors, numbers are represented as integers—one or several bytes stringed together. A 4byte value comprising 32bits can hold a relatively large range of numbers, 2^{32} to be specific. The 32bits can represent the numbers 0 to 4,294,967,295 or -2,147,483,648 to +2,147,483,647. A 32bit processor is designed such that basic arithmetic operations on 32bit integer numbers can be completed in just a few clock cycles. Moreover, with some performance overhead, a 32bit CPU can also support operations on 64bit numbers. The largest value that can be represented by 64bits is really astronomical: 18,446,744,073,709,551,615. In fact, if a Pentium processor could count 64bit values at a frequency of 2.4GHz, it would take it 243 years to count from zero to the maximum 64bit integer.

You would think that integers worked well, but that is not always the case. The problem with integers is the lack of dynamic range and rounding errors.

The quantization introduced through a finite resolution in the number format distorts the representation of the signal. However, as long as a signal uses the dynamic range, this distortion may be considered negligible.

Figure 1 shows what a quantized signal looks like for large and small dynamic swings. Clearly, with the smaller amplitude, each quantization step is bigger relative to the signal swing and introduces higher distortion or inaccuracy.

Calculation gone bad

The following example illustrates how integer math can mess things up.

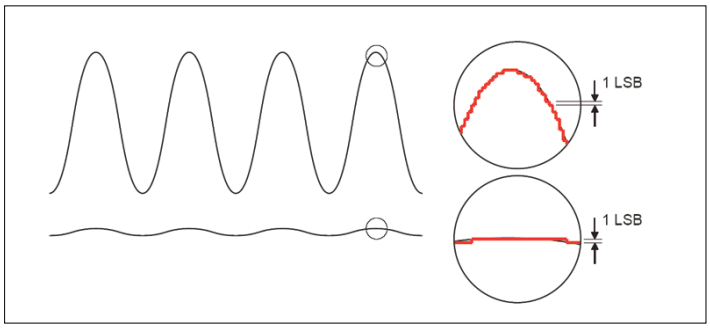

Figure 1: With the smaller amplitude, each quantization step is bigger relative to the signal swing and introduces higher distortion or inaccuracy.

An electronic motor control measures a spinning motor's velocity, which typically has a range of 0-10,000rpm. The value is measured using a 32bit counter. To allow some overflow margin, assume that the measurement is scaled so that 15,000rpm represents the maximum 32bit value, 4,294,967,296. If the motor is spinning at 105rpm, this value corresponds to the number 30,064,771

```
void user_fmul(float *op1, float *op2, float *res)
{
FPU_operand1=*op1;              /* write operand a to FPU */
FPU_operand2=*op2;              /* write operand b to FPU */
FPU_operation=MUL;              /* tell FPU to multiply */
while (!(FPU_stat & FPUready));          /* wait for FPU to finish */
*res = FPU_result               /* return result */
}
```

Listing 1: If you were to connect an FPU to the processor bus, FPU access would occur through specifically designed driver routines.

within 0.0000033 percent, which you would think is accurate enough for most practical purposes.

Assume that the motor control is instructed to increase motor velocity by 0.015 percent of the current value. Because we are operating with integers, multiplying with 1.0015 is out of the question, as is multiplying by 10,015 and dividing by 10,000, since the intermediate result will cause overflow.

The only option is to divide by integer 10,000 and multiply by integer 10,015. Doing that will lead to a value of 30,094,064, but the correct answer is 30,109,868.

Because of the truncation that happens when you divide by 10,000, the resulting velocity increase is 10.6 percent smaller than what you asked for. An error of 10.6 percent of 0.015 percent may not be something to worry about. As you continue to perform similar adjustments to the motor speed, however, these errors will almost certainly accumulate and become a problem.

To overcome this problem, you need a numeric computer representation that represents small and large numbers with equal precision. This is what floating-point arithmetic does.

To the rescue

Floating-point arithmetic is important in industrial applications like motor control and in a variety of other applications. An increasing number of applications that traditionally have used integer math are turning to floating-point representation.

A floating-point number representation on a computer uses something similar to a scientific notation with a base and an exponent. A scientific representation of 30,064,771 is 3.0064771 x 10^7, while 1.001 can be written as 1.001 x 100.

In the first example, 3.0064771 is called the mantissa, 10 the exponent base and 7 the exponent.

IEEE standard 754 specifies a common format for representing floating-point numbers in a computer. Two grades of precision are defined: single precision and double precision. The representations use 32bits and 64bits, respectively.

In IEEE 754 floating-point representation, each number comprises three basic components: sign, exponent and mantissa. To maximize the range of possible numbers, the mantissa is divided into a fraction and leading digit.

The sign bit simply defines

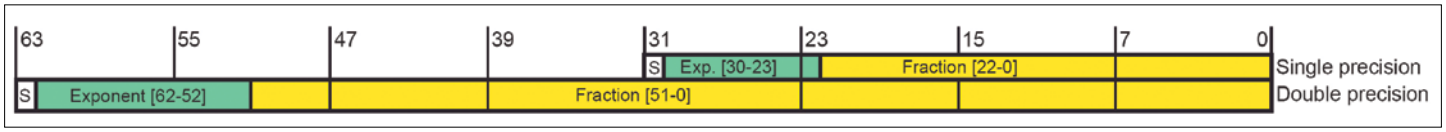

Figure 2: Two grades of precision are defined: single precision and double precision. The representations use 32bits and 64bits, respectively.

the polarity of the number. A value of zero means the number is positive, while a 1 denotes a negative number. The exponent represents a range of numbers, positive and negative. Thus, a bias value must be subtracted from the stored exponent to yield the actual exponent. The single-precision bias is 127, and the double-precision bias is 1,023. This means that a stored value of 100 indicates a single-precision exponent of -27. The exponent base is always 2; this implicit value is not stored.

For both representations, exponent representations of all 0s and all 1s are reserved and indicate special numbers:

- Zero-all digits set to 0, sign bit can be either 0 or 1;
- ±∞-exponent all 1s, fraction all 0s;
- Not a Number (NaN)—exponent all 1s, non-zero fraction. Two versions of NaN are used to signal the result of invalid operations such as dividing by zero, and indeterminate results such as operations with non-initialized operand(s).

The mantissa represents the number to be multiplied by 2 raised to the power of the exponent. Numbers are always normalized, represented with one non-zero leading digit in front of the radix point. In binary math, 1 is the only non-zero number. Thus, the leading digit is always 1, allowing us to leave it out and use all the mantissa bits to represent the fraction (the decimals).

```
float x, y, z;
user_fmul (&x, &y, &z);
```

Listing 2: This will call the driver function in Listing 1 to do the operation z = x*y in the main program.

The binary integer representation of 30,064,771 is 1 1100 1010 1100 0000 1000 0011. This can be written as $1.11001010110000001 0000011 \times 2^{24}$. The leading digit is omitted; the fraction, the string of digits following the radix point, is 1100 1010 1100 0000 1000 0011. The sign is positive and the exponent is 24 decimal. Adding the bias of 127 and converting to binary yield an IEEE 754 exponent of 1001 0111.

Putting all of the pieces together, the single representation for 30,064,771 is shown in **Figure 3**.

Gain some, lose some

You lose the least significant bit of value 1 from the 32bit integer representation because of the limited precision for this format.

The range of numbers that can be represented with single-precision IEEE 754 representation is $\pm(2\text{-}2^{-23}) \times 2^{127}$ or approximately ±1038.53. This range is astronomical compared to the maximum range of 32bit integer numbers, which by comparison is limited to around $\pm 2.15 \times 10^{9}$. Also, while the integer representation cannot represent values between 0 and 1, single-precision floating point can represent values down to $\pm 2^{-149}$, or $\pm \sim 10^{-44.85}$. Is 32bit a much more convenient way to represent numbers? The answer depends on the requirements.

This may be a more convenient way because in the example of multiplying 30,064,771 by 1.001, the result will be accurate.

This may not be the convenient way at other times, though. The number 30,064,771 is not precisely represented. In fact, 30,064,771 and 30,064,770 are represented by the exact same 32bit bit pattern. Thus, a software algorithm will treat the numbers as identical. Much worse, if you increment either number by one a billion times, none of them will change. By using 64 bits and representing the numbers in double-precision format, that particular example could be made to work, but even double-precision representation will face the same limitations once the numbers get big or small enough.

Moreover, most embedded processor cores' ALUs only support integer operations, leaving floating-point operations to be emulated in software. This severely affects processor performance. A 32bit CPU can add two 32bit integers with one machine-code instruction. However, a library routine including bit manipulations and multiple arithmetic operations is needed to add two IEEE single-precision floating-point values.

With multiplication and division, the performance gap just increases. Thus, for many applications, software floating-point emulation is not practical.

Co-processor units

PCs based on the Intel 8086 or 8088 processor came with the option of adding a floating-point coprocessor unit (FPU), the 8087. Although a compiler switch, you could tell the compiler that an 8087 was present in the system. Whenever the 8086 encountered a floating-point operation, the 8087 would take over, do the operation in hardware and present the result on the bus.

Hardware FPUs are complex logic circuits. In the 1980s, the cost of the additional circuitry was significant. Thus, Intel decided that only those who needed floating-point performance would have to pay for it. The FPU was kept as an optional discrete solution until the introduction of the 80486, which came in two versions—one with and one without an FPU. With the Pentium family, the FPU was offered as a standard feature.

Today, applications using 32bit embedded processors with far less processing power than a Pentium also require floating-point math. Other applications benefiting from FPUs are process and automotive control, navigation, image processing, CAD tools and 3D computer graphics including games.

As floating-point capability becomes more affordable and popular, applications that traditionally have used integer math turn to floating-point representation. Examples include high-end audio and image processing. The latest version of Adobe Photoshop supports image formats where each color channel is represented by a floating-point number rather than the usual integer representation. The increased dynamic range fixes some problems inherent in integer-based digital imaging.

If you have taken a picture of a person against a bright blue sky, you know that without a powerful flash you are left with two choices: a silhouette of the person against a blue sky or a detailed face against a washed-out white sky. A floating-point image format partly solves this problem, as it makes it possible to represent subtle nuances in a picture with a wide range in brightness.

Operation CPU	Cycles without FPU	CPU cycles with CPU	Acceleration
Addition	400	6	67x
Subtraction	400	6	67x
Division	750	30	25x
Multiplication	400	6	67x
Comparison	450	3	150x

Table 1: Every basic floating-point operation is accelerated by a factor of 25-150 with an FPU.

Compared to software emulation, FPUs can speed up floating-point math operations by a factor of 20 to 100, depending on the type of operation. However, the availability of embedded processors with on-chip FPUs is limited. Although this feature is becoming increasingly more common at the higher end of the performance spectrum, these derivatives often come with an extensive selection of advanced peripherals and very high-performance processor cores; these are features and performance that you have to pay for even if you only need the floating-point math capability.

Embedded processors

Xilinx's MicroBlaze 4.00 processor makes an optional single-precision FPU available. Users have the choice of whether to spend some extra logic to achieve real floating-point performance or to do traditional software emulation and free up some 20 percent to 30 percent of logic for other functions.

A soft processor without hardware support for floating-point math can be connected to an external FPU implemented on an FPGA. Similarly, any MCU can be connected to an external FPU. However, unless you take special considerations on the compiler side, you cannot expect seamless cooperation between the two.

C-compilers for CPU architecture families that have no floating-point capability will always emulate floating-point operations in software by linking in the necessary library routines. If you were to connect an FPU to the processor bus, FPU access would occur through specifically designed driver routines such as in **Listing 1**.

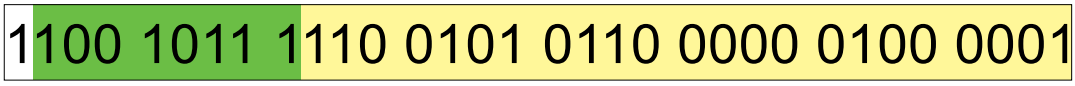

Figure 3: Putting all of the pieces together, the single representation for 30,064,771 is shown in the figure.

To do the operation, z = x*y in the main program, you would have to call the driver function in **Listing 1** with **Listing 2**.

For small and simple operations, this may work reasonably well. For complex operations involving multiple additions, subtractions, divisions and multiplications, such as a proportional integral derivative algorithm, this approach has three major drawbacks:

```
float x, y, z;
y = x * z;
```

Listing 3: The compiler will use those special instructions to invoke the FPU and perform the operation.

- The code will be hard to write, maintain and debug;
- The overhead in function calls will severely decrease performance;
- Each operation involves at least five bus transactions; as the bus is likely to be shared with other resources, this not only affects performance, but the time needed to perform an operation will be dependent on the bus load in the moment.

The MicroBlaze way

The optional MicroBlaze soft processor with FPU is a fully-integrated solution with an FPU operation completely transparent to the user.

When you build a system with an FPU, the development tools automatically equip the CPU core with a set of floating-point assembly instructions known to the compiler.

To perform y = x*y, you would simply write **Listing 3**. The compiler will then use those special instructions to invoke the FPU and perform the operation.

This is simpler. In addition, a hardware-connected FPU guarantees a constant number of CPU cycles for each floating-point operation. Finally, the FPU provides an extreme performance boost. Every basic floating-point operation is accelerated by a factor 25-150 (**Table 1**).

Floating-point arithmetic is necessary to meet precision and performance requirements for an increasing number of applications. Today, most 32bit embedded processors that offer this functionality are derivatives at the higher end of the price range. The MicroBlaze soft processor with FPU can be a cost-effective alternative to ASSP products.