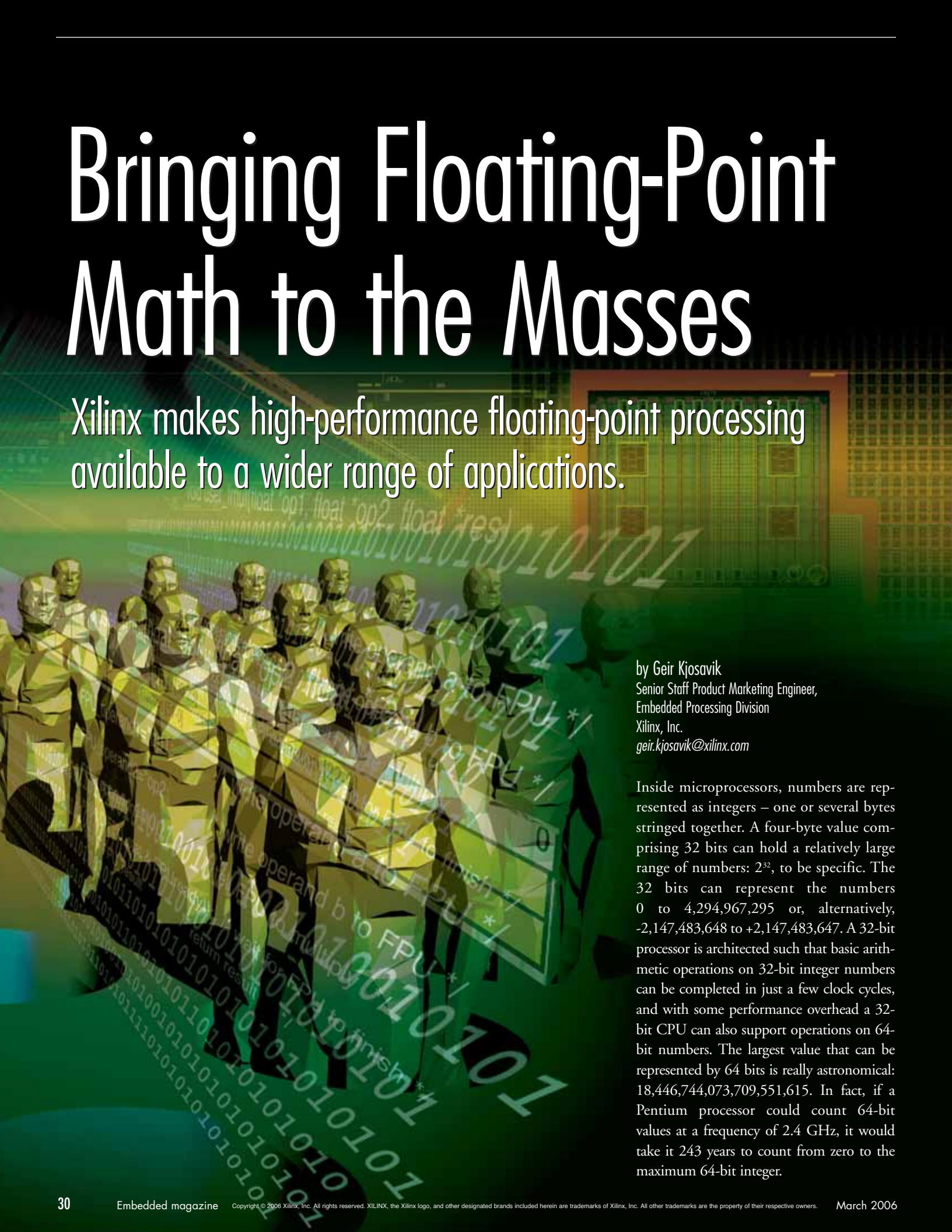
Bringing Floating-Point Math to the Masses

Xilinx makes high-performance floating-point processing available to a wider range of applications.



by Geir Kjosavik
Senior Staff Product Marketing Engineer,
Embedded Processing Division
Xilinx, Inc.
geir.kjosavik@xilinx.com

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

 Copyright © 2006 Xilinx, Inc. All rights reserved. XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

Dynamic Range and Rounding Error Problems

Considering this, you would think that integers work fine, 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 is utilizing the range of numbers that can be represented by integer numbers, also known as the dynamic range, this distortion may be negligible.

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

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

A Calculation Gone Bad

An electronic motor control measures the velocity of a spinning motor, which typically ranges from 0 to10,000 RPM. The value is measured using a 32-bit counter. To allow some overflow margin, let's assume that the measurement is scaled so that 15,000 RPM represents the maximum 32-bit value, 4,294,967,296. If the motor is spinning at 105 RPM, this value corresponds to the number 30,064,771 within 0.0000033%, 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% 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 – because the intermediate result will cause overflow.

The only option is to divide by integer 10,000 and multiply by integer 10,015. If you do that, you end up with 30,094,064; but the correct answer is 30,109,868.

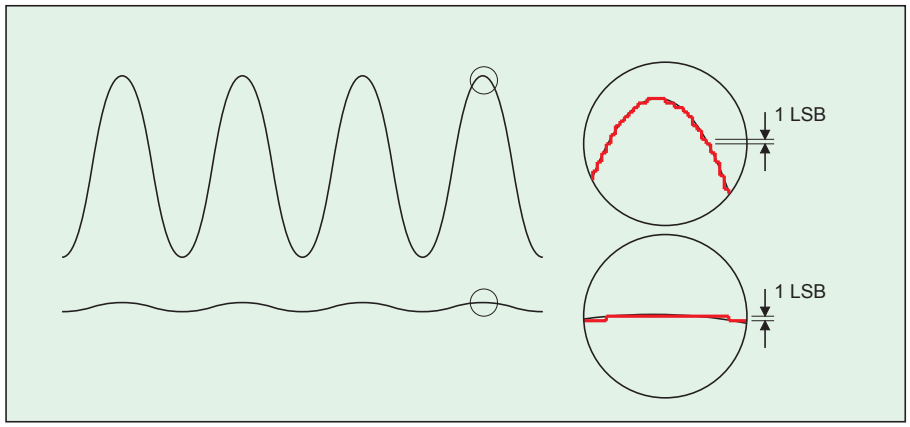


Figure 1 – Signal quantization and dynamic range

Because of the truncation that happens when you divide by 10,000, the resulting velocity increase is 10.6% smaller than what you asked for. Now, an error of 10.6% of 0.015% may not sound like anything to worry about, but as you continue to perform similar adjustments to the motor speed, these errors will almost certainly accumulate to a point where they become a problem.

What you need to overcome this problem is a numeric computer representation that represents small and large numbers with equal precision. That is exactly what floating-point arithmetic does.

Floating Point to the Rescue

As you have probably guessed, floating-point arithmetic is important in industrial applications like motor control, but also in a variety of other applications. An increasing number of applications that traditionally have used integer math are turning to floating-point representation. I'll discuss this once we have looked at how floating-point math is performed inside a computer.

IEEE 754 at a Glance

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 3.0064771×10^7, whereas 1.001 can be written as 1.001×10^0.

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 32 and 64 bits, respectively. This is shown in Figure 2.

In IEEE 754 floating-point representation, each number comprises three basic components: the sign, the exponent, and the mantissa. To maximize the range of possible numbers, the mantissa is divided into a fraction and leading digit. As I'll explain, the latter is implicit and left out of the representation.

The sign bit simply defines the polarity of the number. A value of zero means that the number is positive, whereas 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, and this implicit value is not stored.

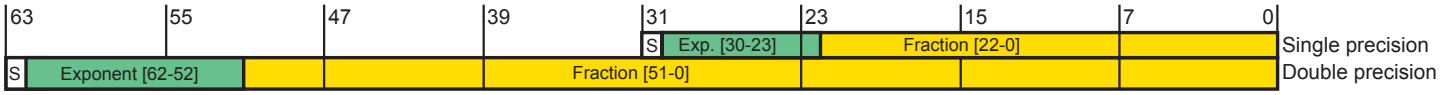


Figure 2 – IEEE floating-point formats

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
- $\pm\infty$: 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; that is, represented with one non-zero leading digit in front of the radix point. In binary math, there is only one non-zero number, 1. 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).

Following the previous number examples, here is what the single precision representation of the decimal value 30,064,771 will look like:

The binary integer representation of 30,064,771 is 1 1100 1010 1100 0000 1000 0011. This can be written as $1.11001010110000001000001 1 \times 2^{24}$. The leading digit is omitted, and 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 yields 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

Notice that you lose the least significant bit (LSB) of value 1 from the 32-bit integer representation – this is because of the limited precision for this format.

The range of numbers that can be represented with single precision IEEE 754 representation is $\pm(2-2^{-23}) \times 2^{127}$, or approximately $\pm 10^{38.53}$. This range is astronomical compared to the maximum range of 32-bit integer numbers, which by comparison is limited to around $\pm 2.15 \times 10^{9}$. Also, whereas the integer representation cannot represent values between 0 and 1, single-precision floating-point can represent values down to $\pm 2^{-149}$, or $\pm \sim 10^{-44.85}$. And we are still using only 32 bits – so this has to be a much more convenient way to represent numbers, right?

The answer depends on the requirements.

- Yes, because in our example of multiplying 30,064,771 by 1.001, we can simply multiply the two numbers and the result will be extremely accurate.
- No, because as in the preceding example the number 30,064,771 is not represented with full precision. In fact, 30,064,771 and 30,064,770 are represented by the exact same 32-bit bit pattern, meaning that a software algorithm will treat the numbers as identical. Worse yet, if you increment either number by 1 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.
- No, because most embedded processor cores ALUs (arithmetic logic units) only support integer operations, which leaves floating-point operations to be emulated in software. This severely affects processor performance. A 32-bit CPU can add two 32-bit 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.

1100 1011 1110 0101 0110 0000 0100 0001

Figure 3 – 30,064,771 represented in IEEE 754 single-precision format

Floating Point Co-Processor Units

For those who remember PCs based on the Intel 8086 or 8088 processor, they came with the option of adding a floating-point coprocessor unit (FPU), the 8087. Though 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, and 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.

Floating Point is Gaining Ground

These days, applications using 32-bit embedded processors with far less processing power than a Pentium also require floating-point math. Our initial example of motor control is one of many – other applications that benefit from FPUs are industrial process control, automotive control, navigation, image processing, CAD tools, and 3D computer graphics, including games.

As floating-point capability becomes more affordable and proliferated, applications that traditionally have used integer math turn to floating-point representation. Examples of the latter include high-end audio and image processing. The latest version of Adobe Photoshop, for example, 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.

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 – MicroBlaze floating-point acceleration

If you have ever 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.

Compared to software emulation, FPUs can speed up floating-point math operations by a factor of 20 to 100 (depending on type of operation) but 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 – features and performance that you have to pay for even if you only need the floating-point math capability.

FPUs on Embedded Processors

With the MicroBlaze™ 4.00 processor, Xilinx makes an optional single precision FPU available. You now have the choice whether to spend some extra logic to achieve real floating-point performance or to do traditional software emulation and free up some logic (20-30% of a typical processor system) for other functions.

Why Integrated FPU is the Way to Go

A soft processor without hardware support for floating-point math can be connected to an external FPU implemented on an FPGA. Similarly, any microcontroller 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 this one:

```
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     */
}
```

To do the operation, z = x*y in the main program, you would have to call the above driver function as:

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

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

- 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 that offers high performance, deterministic timing, and ease of use. The FPU operation is 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:

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

and the compiler will use those special instructions to invoke the FPU and perform the operation.

Not only is this simpler, but 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 to 150, as shown in Table 1.

Conclusion

Floating-point arithmetic is necessary to meet precision and performance requirements for an increasing number of applications.

Today, most 32-bit 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, and results show that with the correct implementation you can benefit not only from ease-of-use but vast improvements in performance as well.

For more information on the MicroBlaze FPU, visit *www.xilinx.com/ipcenter/processor_central/microblaze/microblaze_fpu.htm.*